Transparent Server Shutdown in a Wide-Area Location Service

Spyros Voulgaris, Aline Baggio, Gerco Ballintijn, Maarten van Steen
{spyros, baggio, gerco, steen}@cs.vu.nl

Abstract

A location service provides the means of keeping location info on objects. In a Wide Area Network, the location service is often a critical part and is likely to be implemented as a collection of servers distributed across multiple hosts. In such a collection adding and removing servers is complicated if the location information is partitioned among the servers instead of being replicated and if no service interruption can be tolerated. In this paper we are interested in shutting down and removing a server from such a collection. The solution we propose takes care of redistributing and transferring location data to remaining servers. It handles incoming clients' requests during the termination process and removes the server from the collection. This paper shows that a server can gracefully shutdown, while guaranteeing continuous availability of the location service.

1 Introduction

Services in distributed systems are often implemented by means of multiple cooperating servers. A potential problem in such systems is that changes in the configuration of the set of servers may require temporary deactivation of the service. When high availability is of vital importance, special techniques need to be applied to allow adding or removing servers without disrupting the service.

A location service is an important component of many distributed systems. It provides the means of keeping location information on objects while they move between locations. In a wide-area network, a location service is likely to be implemented by means of a collection of servers that is distributed across multiple hosts. Managing such a collection can be quite complex, especially when there are many servers and continuity of the service is required, even in the presence of adding and removing servers.

This paper deals with the process of removing a server from a set of servers that implements a worldwide location service. We discuss various techniques that allow the server to distribute its content to remaining servers before shutting down permanently. We focus on the Globe Location Service.

The remainder of this paper is organized as follows. Section 2 gives background on the Globe Location Service, which acts as a reference point for our discussion. Section 3 describes the problem of gracefully terminating servers. Section 4 provides a detailed discussion on the issues of redistributing partitioned data and handling clients' requests. It also gives an implementation design. Section 5 analyzes our solution in terms of concurrency of shutdowns and fault tolerance to server crashes. Section 6 compares our solution to approaches taken in the related work. Finally, section 7 outlines our conclusions and presents future work.

2 Background on the Globe Location Service

2.1 The Globe Location Service in Brief

Globe is a large-scale distributed system designed to support trillions of objects. One critical part is the *Globe Location Service* (*GLS*) [2,6], which is responsible for tracking and locating mobile and possibly replicated objects. Every object is assigned a unique *object identifier* and can be accessed at its *contact addresses*, which are the physical locations of its replicas (e.g. IP addresses and port numbers). GLS is responsible for resolving an object identifier into a contact address.

GLS is organized as a hierarchical structure of *domains*, as shown in Figure 1. The top level domain spans the entire network. Each domain *D* may be partitioned into smaller child domains, turning *D* into their parent domain. A lowest-level domain typically corresponds to a campus or a city.

Each domain is represented by at least one *Location Server* (*LS*), shown as a circle in Figure 1. In many cases, several LSs compose a domain and are jointly responsible for providing storage capacity and for handling requests for that domain. For example, we anticipate that the top-level domain, which needs to keep

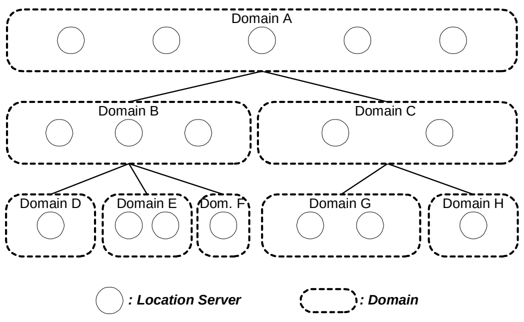

Figure 1: The domain hierarchy in GLS

location information on *each* object in the whole network, will probably be composed of thousands of LSs.

Each domain D keeps location information on objects that reside in the area covered by D. This information is either the object's *contact address* or a pointer, we call *forwarding pointer*, that refers to a child domain. Location information is stored by a domain's LSs in a structure called *contact record* (*CR*). For each object in a domain, there is exactly one CR. This means that the location information is partitioned among LSs.

2.2 The Operations

Information about objects can be retrieved or set using *lookup*, *insert* and *delete* requests. The algorithmic details of these operations have been published elsewhere [2,3,4,5,6]. However, to understand the problem of removing a LS from a domain, we need to explain the essence of these operations:

Lookup: Given the identifier for an object *O* a lookup request returns a contact address for O, if found. When receiving a lookup request, a LS *S* having a CR for O that contains a contact address returns it to the requester. Instead, if the CR for O contains a forwarding pointer, the request is propagated to a LS in the corresponding child domain. Finally if S has no CR for O, it propagates the request to a LS in its parent domain. In the worst case, a lookup request for O is first forwarded to higher-level domains, up until a CR is found for this object. Then the request follows a path of forwarding pointers down to the LS where the contact address is stored.

Insert: An insert request is initiated at one of the LSs of a lowest-level domain and eventually results in storing a new contact address. The operation consists of two phases. An upward phase recursively propagates the insert request up in the domain hierarchy until it reaches either the first domain that already contains a CR for that object or the root. A downward phase installs the contact address and forwarding pointers at the appropriate levels, completing the recursion. The insert operation in a LS of a domain D cannot complete before the request to D's parent domain returns.

Delete: The delete operation is carried out when a contact address is to be removed. The operation is recursively propagated upwards through the domain hierarchy until it reaches either a domain that covers the lowest-level domain where the object also resides, or the root.

Insert and delete operations are together called *update* operations. Lookup or update requests can be submitted by LSs that belong either to the parent or to a child domain. The recursion from level to level is implemented using RPCs. The execution of a RPC usually involves performing another RPC at the parent, thus leading to a chain of RPCs from the leaf upwards, possibly to the root.

In this paper we use the term *client* to denote a LS that submits a request to a LS from another domain. Since a domain generally has many LSs, each one storing a fraction of the domain's CRs, the client has to know exactly to which LS of a domain it should send its request. For that reason, LSs keep mapping information that tells which object identifier is associated to which LS in a given domain.

The mapping information is locally available at the LSs. Each LS has a mapping table per domain. The management of this mapping information and its distribution to each LS is achieved through a *configuration service*. The configuration service makes sure that the mapping information is replaced in an atomic fashion. The design of the configuration service itself could be either centralized or distributed, but this is out of the scope of this paper.

3 The Problem

The algorithms for GLS operations implicitly assume a stable set of LSs. We are interested in making this set more dynamic by adding or removing servers. This paper concentrates on the latter issue: terminating a LS.

We consider unacceptable switching off GLS for shutting down one of its LSs. Instead we want GLS to remain (almost) continuously available. The process of shutting down a LS should therefore be transparent and fault tolerant. It involves the following three actions:

- The terminating LS should distribute its data to (some of) the remaining LSs of the same domain;
- The clients should update their object-identifier-to-LS mapping information for the domain in question;
- The terminating LS should be guaranteed to terminate and should also be able to deter-

ministically find out when the termination process has completed.

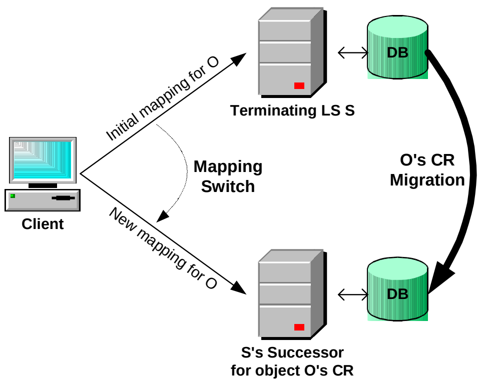

Figure 2: Contact Record Migration and Mapping Switch

Figure 2 shows an object's CR's migration to a new LS and a client's switch to the new mapping. What remains to be defined are the order and the details of these actions, so that it provides seamless service with low overhead in time and resources.

Migrating data between two servers and switching a client's mapping are rather simple problems, if taken independently. The case we are discussing is not so trivial though, due to the following constraints:

- Lookup and update requests keep coming during the termination procedure and a decision has to be made whether they should be handled and by which LS;
- It is not clear when clients should assume that a LS has been removed, i.e. when to change their mapping;
- Updating mapping information is assumed to be atomic: a client learns the mapping to the new set of LSs for a specific domain for *all* objects together, and not on a per object basis. However, different clients are allowed to update their mappings independently.

4 Solutions

4.1 Options and Policies

Assume a LS is currently terminating and needs to distribute its CRs. A few options need to be discussed before describing the actual solutions. First, there are two ways to distribute the terminating LS's CRs to the remaining LSs: pushing and pulling. Pushing means that the terminating LS distributes its CRs to the appropriate LSs. Pulling means that a remaining LS S_1 fetches CRs from the terminating LS S_2. This implies that S_1 is notified of S_2's termination. From now on we will refer to these two policies respectively as *push-CRs* and *pull-CRs*.

Second, the distribution of the new mapping information can be done in two analogous ways: by pushing or pulling. Pushing means that the configuration service sends new mapping information to clients. Pushing can occur at the beginning or at the end of the termination procedure. We call it respectively *push-mapping-at-start* and *push-mapping-at-end*. Pulling means that clients request the new mapping information at their own initiative. We call it *pull-mapping*.

Third, requests arriving at a LS during the termination procedure can be handled in various ways. For instance they can be ignored, rejected, stalled, forwarded, or serviced.

Our analysis towards an optimal solution can be viewed as an effort to fine-tune the options described above and find the best combination.

4.2 Distribution of Contact Records

If the terminating LS uses the push-CRs policy for distributing its CRs, it starts working immediately towards its main goal: to distribute its content to the other LSs. However, if a RPC of an update operation on some CR is still pending, the terminating LS should wait until the operation is completed. Only then can it ship that CR to its new destination. A LS starts shipping CRs from the moment the shutdown procedure starts. The time needed to complete the shutdown depends only on the number of CRs it has to send and the speed of the network connections to the other LSs. In addition to minimizing the distribution time, it is also clear for the terminating LS when the shutdown procedure is completed. Finally, the push-CRs policy gives the terminating LS more control over the process of shipping CRs. For example, it can decide to compress outgoing packets and achieve higher compression rates by putting CRs destined for the same LS together.

With the pull-CRs policy, the terminating LS does not ship any CRs until explicitly requested to. This policy introduces three significant drawbacks. First, the LS's termination is delayed if the other LSs are not pulling CRs in a timely manner. But even if they are, there is no decrease of the distribution time: the number of CRs to be shipped remains the same. Second, the terminating LS looses control over the termination procedure. It has to keep track of which CRs have not yet been shipped and wait until a LS pulls them before it can actually shutdown. Third, requests sent directly to one of the remaining LSs can be additionally delayed by the time it takes to pull the associated CR from the terminating LS. Fetching a CR involves sending a message to the terminating LS, retrieving and formatting the requested CR there, and sending it back to the other LS. Applying some prefetching policy to pull in CRs before they are requested approximates the push-CRs policy, without providing any of its advantages.

For these reasons we consider the push policy as the most appropriate for distributing CRs.

4.3 Handling of Requests and Distribution of New Mapping

Two issues remain to be solved: handling incoming lookup and update requests and distributing new mapping. As we shall see, these two issues are highly related and affect each other. Therefore, we will analyze them in parallel.

The simplest policy is refusing any incoming request during the termination procedure. We can either ignore requests or send explicit rejections to clients. Considering the need for seamless operation, we want clients to learn immediately that their request cannot be serviced. Distribution of the new mapping in conjunction with the rejecting-requests strategy can work as follows. With the *pull-mapping* policy, a client can retrieve the new mapping from the configuration service upon a rejected request. It can then resubmit the request to the new LS. With the *push-mapping-at-start* policy, the client sends the request directly to the new LS. In either case the request is serviced only after the associated CR has been shipped to the new LS. With the *push-mapping-at-end* policy, the request is not serviced until the termination is completed. Only then does the client learn about the new mapping and sends the request to the new LS. In all three cases of mapping distribution, there is a period during which requests cannot be serviced. This does not satisfy our requirement for seamless operation. Therefore, rejecting requests during termination cannot be a solution.

A different approach is having the terminating LS service incoming requests until CR shipping is over. While the LS is shipping its CRs to other LSs, it still has copies of them in its local storage area as well. To maintain consistency, the results of update requests for CRs that have already been shipped need to be propagated as well to their new LSs. Considering that the LS has to wait for all pending updates before it can terminate, this may put it in an indefinite waiting status: new update requests for shipped CRs can keep arriving. Thus this approach is inappropriate for our situation. Indefinite waiting can be avoided by applying a time limit, after which by applying a time limit, after which the LS rejects all incoming requests in order to complete migration of the CRs that are still awaiting to be shipped. However this approach renders the service unavailable for some period and is therefore inappropriate as a solution.

Another approach that is a tradeoff between the previous two is to temporarily and selectively service incoming requests. The terminating LS starts by servicing requests for all objects, but gradually stops servicing requests for objects whose CRs have been shipped. The question is *when* to stop servicing requests and for *which* objects. As long as an object's CR has not been shipped, all associated requests are serviced. In addition the terminating LS also services lookup requests for shipped CRs that have not been updated since their shipment. Update requests for already shipped CRs are rejected in order not to break consistency with the CRs' new LSs.

With this request handling, mapping distribution works as follows. Using the *push-mapping-at-start* policy, clients would not take advantage of the terminating LS's ability to service requests while CRs are being shipped. No requests would be serviced neither by the terminating server nor by the new one before the completion of the termination. Using the *push-mapping-at-end* policy, a request would be serviced only if its associated CR has either not been shipped yet or not been updated since it was shipped. Using the *pull-mapping* policy, a client can retrieve the new mapping when its first request gets rejected by the terminating LS and resubmit it immediately to the new LS. However, getting the new mapping has an impact on the client's subsequent requests. Requests concerning CRs remapped to the new LS are directly sent to it. As a consequence, a request regarding a non-shipped CR will be stalled at a new LS until the CR is shipped. This renders the service unavailable for some CRs for a period of time. This cannot be an acceptable solution.

4.4 Advanced Solution

Based on the discussion in the previous section, we suggest the following solution for providing continuously available service with low overhead in bandwidth, delays and processing power, and a timely and guaranteed completion of the LS's termination. Our solution employs the *push-CRs* policy for distributing CRs to other LSs and the *push-mapping-at-end* for distributing the new mapping to the clients, pushing CRs first and mapping immediately after CR distribution has completed. In between the beginning of CR distribution and its completion, the terminating LS takes on the role of a proxy. It forwards the requests for already shipped CRs to the appropriate LSs, gets the replies and forwards them back to the clients. Thus, CR migration is still transparent to clients until it is completed.

The scenario for this solution is as follows. A terminating LS *S* informs the configuration service about its intention to shut down and obtains the new mapping information. Based on it, S starts shipping CRs to the appropriate LSs, while still serving incoming requests for the CRs that have not been shipped yet. Even after a CR has been shipped, S keeps servicing lookup requests for it for as long as it is still consistent with the copy at the new LS. This means until the first update request for that CR is received by the terminating LS. Upon reception of an update request for a shipped CR, S starts acting as a proxy between any client and the new LS for all subsequent requests for that CR. The new mapping is finally pushed to all the clients when all CRs have been readily distributed to the new LSs, avoiding a noticeable interruption in the service.

4.5 Implementation Design

For implementing the advanced solution, we associate a *status* with each CR of the terminating LS. The status takes one of the following values: *LOCAL* when the CR has not been shipped yet, *BEING UPDATED* when an update request is pending for that CR, *SHIPPED* when it has been shipped, and *NOT IN SYNC* when it has been updated after having been shipped.

Figure 3 shows the state diagram executed by the terminating LS. All CRs are initially assigned the status LOCAL. Upon reception of an update request, the associated CR's status is changed to BEING UPDATED. It remains in this state as long as the update's RPC is pending. During that time, the CR cannot be shipped. When the update is completed, the CR's status is set back to LOCAL. Once LOCAL CRs have been shipped to their new LS, their status is changed to SHIPPED. Lookup requests are serviced when the associated CR's status is LOCAL, BEING UPDATED or SHIPPED. If an update request comes for a SHIPPED CR, the CR's status is changed to NOT IN SYNC, the final state of the state diagram. From this point on, all requests for that CR are forwarded to the new LS.

We are considering a multithreaded model as the most appropriate to implement the logic described above. Figure 4 shows the pseudocode for our threads. The *CR Distribution Thread* is started at the beginning of the shutdown procedure. It is devoted to shipping LOCAL CRs to their new LSs, and marking

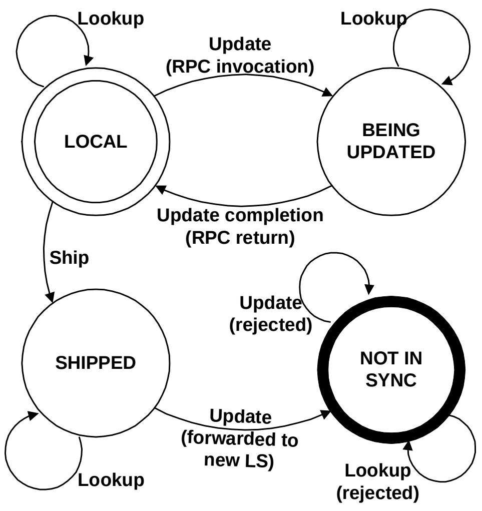

Figure 3: State diagram for CRs

them as SHIPPED. A separate thread, the *Request Handling Thread* is spawned for each incoming request. It services the request locally if it is possible. Otherwise it submits the request to the associated CR's new LS, waits for a reply and sends it back to the client. Several optimizations can be applied in an actual implementation, such as spawning a new thread only for update requests, or keeping a pool of threads to service incoming requests, but this is out of the scope of this paper

5 Further Analysis of Advanced Solution

Let us concentrate on some further features of the advanced solution, namely concurrent shutdowns and fault tolerance to LS crashes.

5.1 Concurrency

In a reasonably sized domain being supported by several thousands of LSs, two or more LSs may need to terminate at the same time. The advanced solution operates flawlessly even in the case of overlapping terminations of multiple LSs.

Let us take a simplified case in which just two LSs, *A* and *B*, terminate during the same time. Assume that A contacts the configuration service first and starts shipping its CRs to the rest of the LSs in its domain. Before A's CR migration is completed, B decides to terminate and contacts the configuration service. B receives the new mapping, which already excludes A, and also starts shipping its own CRs to the remaining LSs of the domain, excluding A. Meanwhile the configuration service pushes the new mapping, which now also excludes B, to A. A ships the remaining of its CRs to the LSs suggested by the latest mapping. Those

```
CR Distribution Thread

while not all CRs have been shipped
     select a CR with status LOCAL
     change CR status to SHIPPED
     ship CR to the appropriate LS
end while
terminate


Request Handling Thread
on LOOKUP request:
     if CR status is LOCAL or BEING UPDATED or SHIPPED
          service request by returning the CR's value stored locally
     if CR status is OUT_OF_SYNC
          submit request to the CR's new LS
          forward reply to the client

on UPDATE request:
     if CR status is LOCAL
          change CR status to BEING UPDATED
          invoke RPC
          change CR status to LOCAL
     if CR status is BEING UPDATED
          wait for current update completion (*)
          change CR status back to BEING UPDATED
          invoke RPC
          change CR status to LOCAL
     if CR status is SHIPPED
          change CR status to NOT IN SYNC
          submit request to the CR's new LS
          forward reply to the client
     if CR status is NOT IN SYNC
          submit request to the CR's new LS
          forward reply to the client

(*) i.e. wait for the status to go back to LOCAL
```

Figure 4: Pseudocode for the termination procedure

CRs of A that were already shipped to B will be re-shipped by B to their new destinations, according to the latest mapping.

By induction, we could prove that this algorithm works for any number of concurrently terminating LSs, provided that at least one of the domain's LSs remains alive.

5.2 Fault Tolerance

Another problem we need to address is fault tolerance. We distinguish between two scenarios of a LS crash. Let *S* be a terminating LS of domain *D*. In the first scenario S crashes before its termination is completed. Assuming that the status of each CR is persistently stored in the LS, it can just resume from the point it crashed.

In the second scenario, another LS *S'* from domain D has crashed, while S should send it some CRs. S can wait until either S' restarts or until the configuration service characterizes it as being unavailable and excludes it from the mapping information. In the latter case, when S receives the new mapping that excludes S', it should redistribute all the CRs that were previously mapped to S' to the remaining LSs.

It is important to note that the existing GLS crash-recovery mechanisms and algorithms [3] can be applied.

6 Related work

The problem that we address in this paper resembles that of transparent failover in distributed systems. When a process fails, a backup process takes over the service provided by the failing process. Such a failover is often implemented by means of a hot standby, effectively introducing replication. These schemes do not apply to our situation, as we seek solutions to distribute data across remaining servers without introducing replication.

A framework that shares some common points with the problem described in this paper is RaDaR [7], an architecture for a global web-hosting service. In RaDaR, participating servers (*hosts*) are grouped in sets called *areas*, each area having its own dedicated server called *redirector*, acting as an index of the object *replicas* hosted by the area's hosts. The redirector also keeps a mapping of the *symbolic name* of an object to the (possibly multiple) *physical addresses* of the hosts in its area where a replica for that object can be found.

Migrating a replica from one host to another is performed by replica creation on the recipient host followed by replica deletion on the source host. After the new replica is created, the area's redirector includes it in the mapping. Then the old replica is first excluded from the mapping and then deleted from the host. Whenever a host wants to terminate, it separately migrates each of the replicas it has and terminates when all of them have been migrated.

This approach is significantly simpler than ours. However, it comprises a solution to a problem for an architecture that is different in three main aspects. First, redirectors store the mapping in a centralized fashion for each area. In our model each client stores the mapping locally. Second, RaDaR's mapping can be updated independently for each replica. which is impossible in our case. The mapping is only updated globally, not on a per-CR basis. Third, an object can be simultaneously mapped to more than one replicas, on different hosts of an area, whereas in our model a contact record is always mapped to exactly one LS of a domain.

7 Conclusions and Future Work

This paper has dealt with a management issue for the Globe Location Service: how to gracefully remove a location server from a domain without disrupting the operation of the service as a whole and ensuring a timely completion of the removal procedure. The solution consists of two parts. First, handing over the terminating location server's contact records to the remaining location servers of the same domain; second, changing the clients' contact-record-to-server mapping information to reflect the new set of location servers. Our solution involves low delays in the servicing of requests during the location server's termination, adds no significant processing to the location servers involved, and terminates in a timely period.

The problem we addressed within the context of GLS can be formulated in terms of a more generic problem. We assume a set of servers storing data records in a distributed fashion. Records are partitioned across these servers. Clients are processes that submit to the servers requests to *read* or *write* records. A request always concerns one record at a time. Read is always executed locally on the server, while write involves blocking communication to an external server. For every request, a client knows exactly which server to contact, based on a locally stored mapping table. The mapping table maps records to servers and is updated by an independent service, the configuration service. The problem is how to shutdown one or more servers, distributing their data to the remaining ones and updating the clients' mapping so that the termination procedure is fully transparent.

A plethora of issues related to management of the location service need to be further explored. This includes dealing with non-graceful location server terminations, research on the architecture of the configuration service, and issues dealing with changes in the domain hierarchy.

References

[1] M. van Steen, P. Homburg, A. S. Tanenbaum, “Globe: A Wide-Area Distributed System”, *IEEE Concurrency*, pages 70-78, January 1999

[2] M. van Steen, F.J. Hauck, G. Ballintijn, A. S. Tanenbaum, "Algorithmic Design of the Globe Wide-Area Location Service", *The Computer Journal*, 41(5):297-310, 1998.

[3] G. Ballintijn, M. van Steen, A. S. Tanenbaum, "Simple Crash Recovery in a Wide-area Location Service", *Proc. 12th Int'l. Conf. on Parallel and Distributed Computing Systems*, Fort Lauderdale, Florida, August 1999, pages 87-93

[4] G. Ballintijn, M. Sandberg, M. van Steen, "Scheduling Concurrent RPCs in the Globe Location Service", *Proc. Third Annual ASCI Conference*, Heijen, The Netherlands, June 1997, pages 28-33

[5] A. Baggio, G. Ballintijn, M. van Steen, "Mechanisms for Effective Caching in the Globe Location Service", *Proc. 9th ACM SIGOPS European Workshop*, Kolding, Denmark, September 2000, pages 55-60

[6] M. van Steen, F. J. Hauck, P. Homburg, and A. S. Tanenbaum, “Locating objects in wide-area systems”, *IEEE Communications Magazine*, pages 104–109, January 1998.

[7] M. Rabinovich, A. Aggarwal, “RaDaR - A Scalable Architecture for a Global Web-Hosting Service”, *The 8th Int. World Wide Web Conf*, May 1999