Practical Aspects of IP Take-Over Mechanisms

Christof FETZER
AT&T Labs – Research
christof@research.att.com

Neeraj SURI
Technical University of Darmstadt
suri@informatik.tu-darmstadt.de

Abstract

Transparent replication has been viewed as the holy grail of fault-tolerant computing. We discuss issues arising when using atomic broadcast to replicate services that are accessed via the Internet. We show how some of these issues can be addressed by the use of IP based replication schemes. In particular, we argue for the replication of services using a simple IP take-over scheme. Also, we show how a simple fail-over scheme can be combined with a load shedding and balancing mechanism.

1 Introduction

The dependability community has been investigating concepts and mechanisms to improve the availability of services. Fault-tolerance mechanisms use replication to increase the availability of a service. Some researchers still view *transparent replication* of services as the holy grail of fault-tolerant computing. *Transparent replication* means that a service can be replicated without the need to modify the client or server code. For example, in recent years a fault-tolerant version of CORBA was defined that enables programmers a more or less transparent replication of servers (which, in this case are represented by objects).

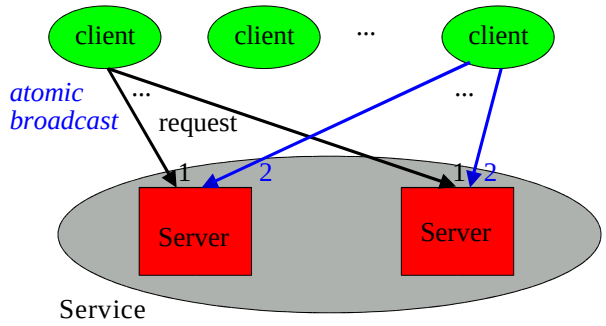

Figure 1: Transparent active replication mechanisms use an atomic broadcast mechanism to replicate the sate of a service.

Transparent replication has been investigated in the context of atomic and causal broadcast mechanisms, and there has been an enormous effort to make broadcast mechanisms very efficient. A client sends a request to a service by broadcasting it to all servers (see Figure 1). The client and the server code can be conventional client/server code in which the communication mechanism is replaced by an atomic broadcast mechanism. An atomic broadcast mechanism ensures that a client request is either delivered to all servers or to none of the servers (atomicity property) and all requests are delivered in the same order to all servers (total order property). Assuming that the server code is deterministic, the use of an atomic broadcast mechanism ensures that all servers stay in sync.

In recent years, researchers have started to argue against the use of transparent replication (e.g., [6]). One argument against replication transparency is that application developers need or want to control how an application is replicated, e.g., they want to control when a fail-over should take place. In this paper, we address various implementation, administrative, and performance issues in the context of *wide-area services*, i.e., services accessed by clients via a wide-area network like the Internet. In particular, we discuss how IP based replication mechanisms can address some of these issues.

2 Transparent Replication Issues

Most clients use TCP/IP or UDP/IP to communicate with a service because of the universal availability of these protocols. To support transparent replication of such IP based services, one could tunnel all TCP and UDP traffic through an atomic broadcast mechanism (see Figure 2). This can be implemented by wrapping the client and server code with tunneling code, i.e., no modification of the client or server code is needed.

Modifying client code and even wrapping client code can be difficult. Clients of wide-area services are typically in a different administration domains than the servers (see Figure 3). Also, the clients themselves can be in many different administration domains. Even if the client code is open source, installing code modifications is not always possible because not all administrative domains might be willing to

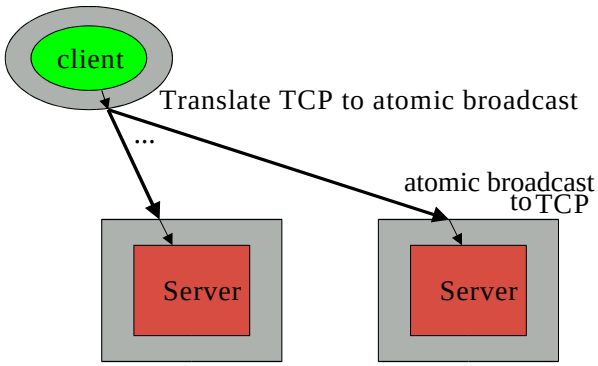

Figure 2: TCP or UDP traffic could be transparently tunneled through an atomic broadcast mechanism with the help of server and client side wrappers.

run the modified version. The same is true for wrappers. Some administrative domains have very strict rules about what software and hence, wrappers, can run on a host.

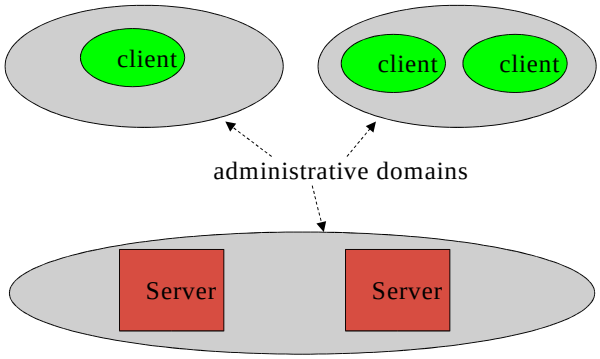

Figure 3: Clients and servers of wide-area services are typically in different administrative domains. Installing modified or wrapped client code can hence be very difficult.

Implementing an efficient atomic broadcast mechanism for wide-area systems is non-trivial. Running atomic broadcast code on routers would increase the performance. However, installing an atomic broadcast code on routers is close to impossible. In wide-area services, the network link between a client and the servers has typically a lower throughput and a higher latency than the links between the servers. To cope with the different link properties – if no protocol code can be run on the routers – it make sense to implement a server side broadcast (see Figure 4).

3 TCP-Based Server Side Broadcast

To avoid the above mentioned administrative and performance issues, one can use a TCP-based server side broadcast mechanism. Recently, a few such mechanisms have been proposed [1, 5, 4, 7]. The main ideas are that (1) clients use an unmodified TCP stack to communicate with the service, and (2) that servers coordinate amongst themselves to make sure they receive the same set of messages.

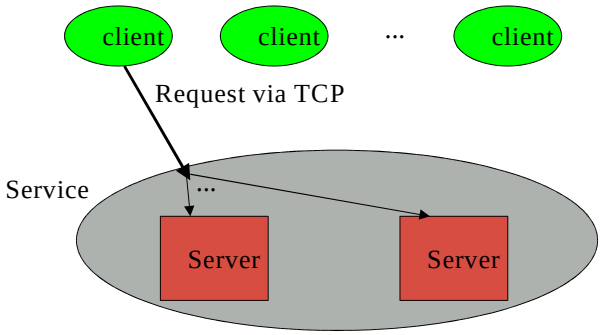

Figure 4: Server side broadcast makes use of the faster links between the servers to improve the performance of broadcasting requests.

Many services are non-deterministic. One reason is the prevalent use of multi-threading to be able to harness the power of multi-processor systems. Another source of non-determinism are resource depletion failures in the operating system that propagate to the application level. Non-deterministic servers can be kept in sync with the help of a leader/follower protocol (see Figure 5).

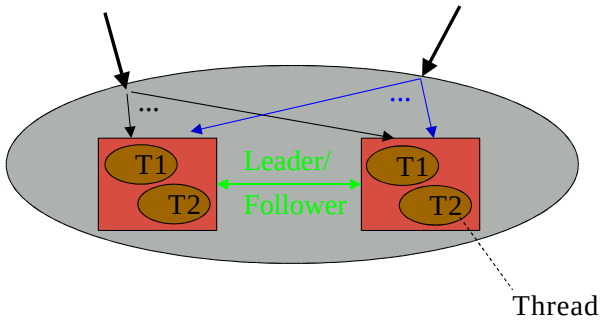

Figure 5: Most services are non-deterministic, e.g., due to the use of multi-threading. To keep servers in sync, one needs a coordination protocol like a leader/follower protocol.

The end-to-end argument implies that there is no need of enforcing a total order on the messages delivered by a TCP-based server side broadcast mechanism. Messages from each client are enqueued at a unique TCP socket. An order on the processing of messages is enforced by the leader / follower protocol. Instead of enforcing a total order on all messages, it is sufficient that a TCP-based server side broadcast mechanism enforces that servers receive the same sequence of messages per client, i.e., per socket (see Figure 6).

TCP-based server side broadcast mechanisms together with a leader / follower protocol can be used to implement transparent replication of services. For example, consider that 1) SAMBA clients do not reconnect to a SAMBA service after the SAMBA server has failed, and 2) it is not possible to change the SAMBA client software. In this case, transparent replication using TCP-based server side broadcast mechanisms is an appropriate approach to increase the availability of the SAMBA service (see Figure 7). In par-

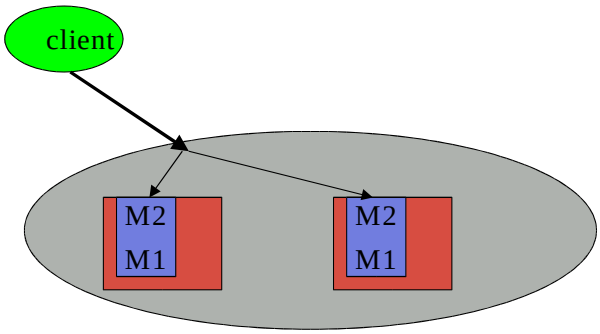

Figure 6: A TCP-based server side broadcast mechanism ensures that correct servers receive the same sequence of messages from a client. There is however no ordering between the messages received from different clients.

ticular, if the SAMBA server is deterministic, this is good solution to cope with host crash failures.

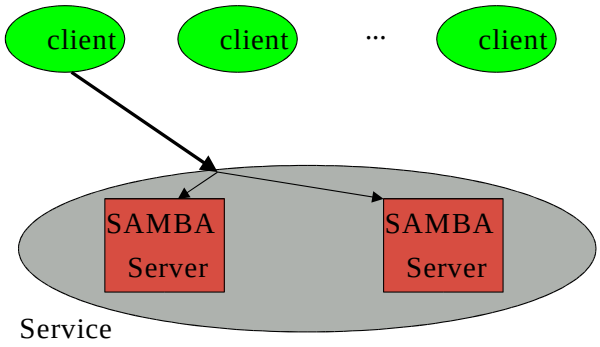

Figure 7: TCP-base server side replication can be used to replicate a SAMBA server transparently. Non-deterministic server code will require a leader/follower protocol to keep the servers in sync.

Implementing a TCP-based server side broadcast mechanism and a leader/follower protocol is non-trivial. Both mechanisms need to be kept up to date with constantly changing program environments. The TCP-based server side broadcast might need to be updated for each new version of the TCP stack. The leader/follower protocol needs to be able to cope with all non-deterministic library functions. Hence, a leader/follower protocol implemented on the library level might need to be updated for each new library version. Keeping the program environment constant is rarely an option because at least security patches need to be applied. A constant update of the program environment makes it very difficult to maintain a robust transparent replication mechanism. Hence, a different replication mechanism might be a better choice.

4 Simple IP Address Take-Over

An alternative to transparent replication, which is actually widely used in industry, is a simple IP address take-over mechanism. In this approach a service is represented by one or more IP addresses. Clients use this service IP address to communicate with a service via UDP or TCP (see Figure 8). Communications in wide-area networks is typically unreliable. Hence, many clients used in WANs retry after the connection to the service has failed.

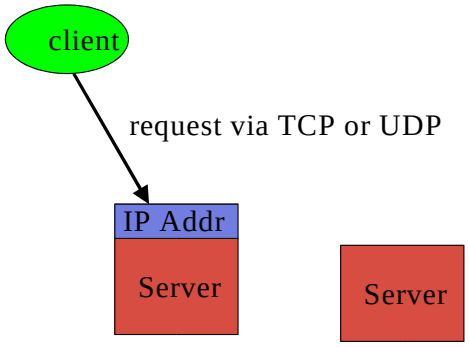

Figure 8: A service is identified by a IP address (and port). If a request fails, many clients retry the request.

Many services running on a single host are automatically restarted when the service crashes. Server crash failures can be masked by clients retrying failed requests. To deal with host crash failures in addition to pure service crash failures, one can restart the service on a different host after the original host failed. In this case, client retries mask host crash failures as long as the new host takes over the service IP address (see Figure 9).

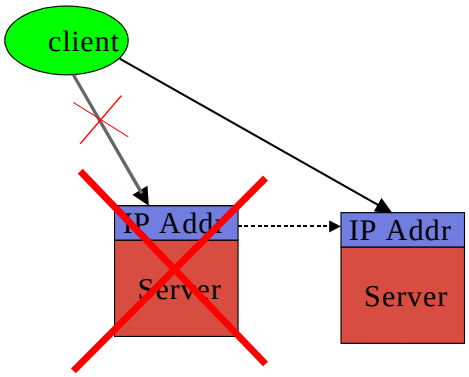

Figure 9: If a host crashes, the server code running on the host crashes. Host crashes can be masked by restarting the server on a new host, assigning the service IP address to this host, and clients retrying failed requests.

Stateless Server Design

To process client requests correctly, most services need to keep state. When a server crashes, it loses its in memory state. The standard way to make a service restartable is to keep sufficient state on a stable storage system (see Figure 10). We call this state the *inter-request state*. Keeping the inter-request state on a stable storage system makes the service *stateless*.

Note that transmission of requests via a wide-area network typically take many milliseconds. Hence, the overhead of storing inter-request state on stable storage might

be acceptable as long as the stable storage system is sufficiently fast. One can use a variety of techniques to make the storage system fast and highly available, e.g., one can use replicated memory storage with battery backup. Note that the storage system needs to be accessible from multiple hosts and should not be a single point of failure.

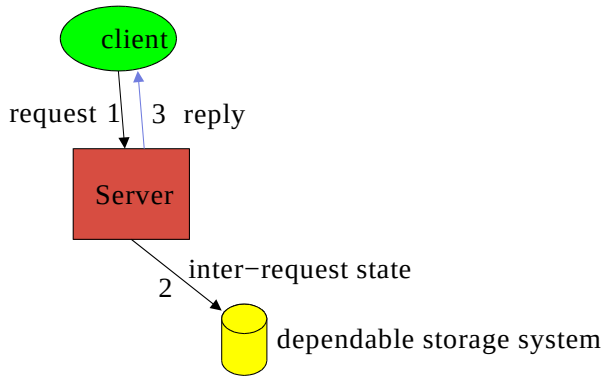

Figure 10: In a stateless service design, the inter-request state is saved on stable storage before replying to a request.

When the host executing the server code crashes, the server code is restarted on a new host. Retried client requests are only processed if there is no reply stored on stable storage. Otherwise, the previous reply is resent.

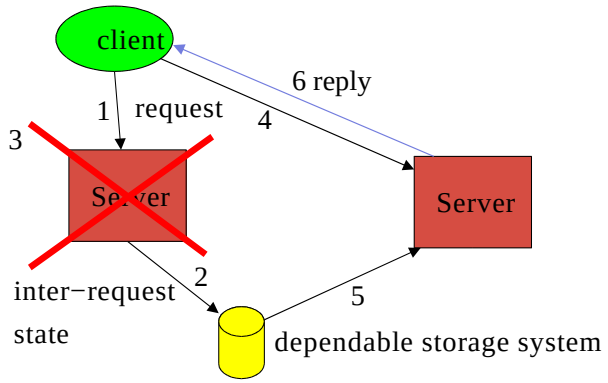

Figure 11: Replies to messages are stored on stable storage and the stored replies are resent when a retried request is received.

Implementing a Simple IP Address Take-Over

We have implemented a simple IP address take-over mechanism. To do this, we derived a highly available leader election protocol from the protocol described in [3]. The idea is that the leader assumes the service IP address (see Figure 12). If a leader is demoted, it must give up the service IP address. This can be enforced by assigning an expiration time to the IP address similarly to the dynamic IP addresses assigned by DHCP.

Our leader election protocol implements a slightly stronger specification than the general leader election specification. The general leader election problem specifies that (1) at any time there is at most one leader, and (2) infinitely often there

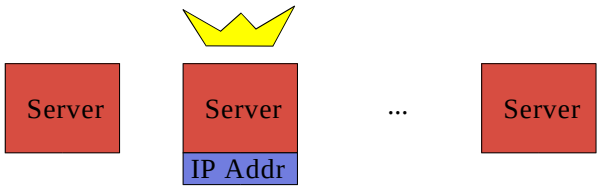

Figure 12: The leader assumes the service IP address while all other hosts have to disable the service IP address.

exists a leader (unless all hosts crash). In most systems one has a preference of which host should run the server code and also one wants to keep the service running on the same host as long as possible. Hence, we extended the specification by the following requirements: (3) the server with the highest priority becomes leader (unless a failure occurs), and (4) only a failure can lead to a leader change.

5 Load Balancing and Fail-Over

For small work-loads, a server running on a single host might be sufficient. For large work-loads, it is often more cost effective to distribute the load over multiple hosts. We introduce a novel load discretization approach that combines load balancing with a simple IP take-over mechanism.

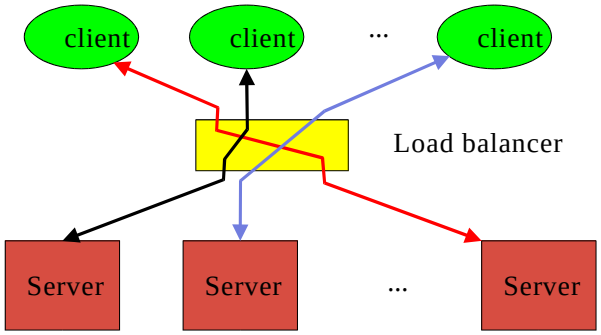

Figure 13: A load balancer box forwards incoming client connections to individual hosts.

Load balancing can be achieved by (1) using a load balancer that forwards TCP connections to a set of servers (see Figure 13), or (2) using a rotating DNS scheme to distributed the load over a set of servers. Our approach can be combined with both existing approaches.

In our approach, we assign a set of IP addresses to the service. Each client request is mapped to exactly one of these IP addresses. This can either be done with the help of a load balancer (see Figure 14) or using a rotating DNS scheme.

To balance the load or to do a fail-over, our approach can reassign IP addresses to servers (see Figure 15). In particular, one server can be assigned more than one IP address, e.g., when a host crash failure forces other servers to take over the IP addresses previously assigned to the crashed server. Note that if too many hosts crash, the remaining

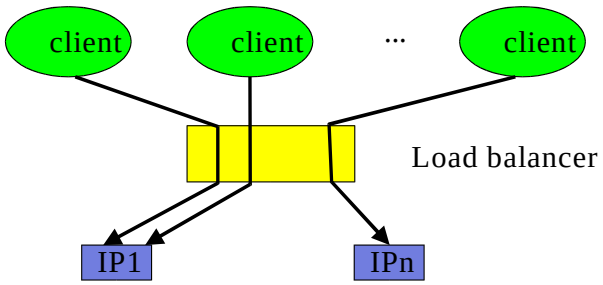

Figure 14: The load balancer (or DNS server) maps client requests to a unique IP address.

servers might not be able to take over all the work load. Hence, one needs to facilitate load shedding. In our approach this is performed by not assigning all service IP addresses to servers.

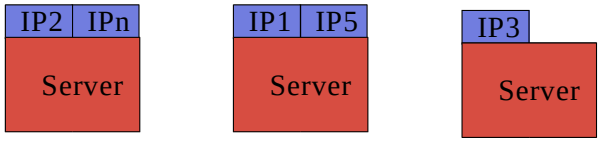

Figure 15: A server can be assigned multiple IP addresses.

Our approach can be implemented with the help of a membership protocol like the one described in [2]. In particular, we need (1) synchronized clocks, (2) an agreement on the current members (i.e., servers), and (3) an agreement on the times at which the membership changes. The assignment of IP addresses to servers can be done by a mapping that, given the current membership view, assigns each server a set of unique IP addresses. Since the change of the membership is synchronized in time, one can avoid that an IP address is assigned to more than one server at a time.

To clarify this approach, consider a system consisting of three servers $S1$, $S2$, and $S3$ and three IP addresses $IP1$, $IP2$, and $IP3$ (see Figure 16). Initially, IP address IPi is assigned to server Si. Let us consider that server $S3$ crashes and at time $T1$ the membership changes from $\{S1, S2, S3\}$ to $\{S1, S2\}$. In this case, server $S1$ takes over the IP address $IP3$ from $S3$. Assume that a server is capable of processing the load arriving via two service IP addresses but not three addresses. If server $S2$ crashes and the membership changes at time $T2$, $S1$ still only servers IP addresses $IP1$ and $IP3$ and address $IP3$ stays unassigned. In this case, a load shedding is performed by leaving a service IP address unassigned.

6 Conclusion

In this paper we argued for the use of IP-based replication mechanisms for wide-area services. Instead of using an atomic broadcast based transparent replication, we discussed reasons for using a TCP-based server side broadcast

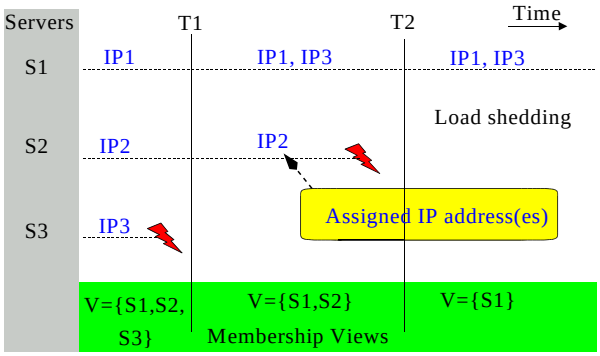

Figure 16: The current membership view can be used to assign the service IP addresses to the servers.

mechanism instead. While IP-based transparent replication can be a good solution, we also argue that stateless server designs combined with IP take-over schemes might in many instances be a more robust design choice. We proposed a novel approach to combine an IP take-over mechanism with a load balancing and load shedding scheme.

References

[1] L. Alvisi, T.C. Bressoud, A. El-Khashab, K. Marzullo, and Z. Zagorodnov. Wrapping server-side tcp to mask connection failures. In *Proceedings of Infocom 2001*, April 2001.

[2] C. Fetzer and F. Cristian. A fail-aware membership service. In *Proceedings of the 16th Symposium on Reliable Distributed Systems*, pages 157–164, Oct 1997.

[3] C. Fetzer and F. Cristian. A highly available local leader service. *IEEE Transactions on Software Engineering*, pages 603–618, Sept.-Oct. 1999.

[4] Shivakant Mishra, Manish Marwah, and Christof Fetzer. Tcp server fault tolerance using connection migration to a backup server. In *International Conference on Dependable Systems and Networks*, San Francisco, CA, USA., June 2003.

[5] M. Orgiyan and C. Fetzer. Tapping tcp streams. In *Proceedings of the IEEE International Symposium on Network Computing and Applications*, February 2002.

[6] Werner Vogels, Robbert van Renesse, and Ken Birman. Six misconceptions about reliable distributed computing. In *In Proc. of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal., September 1998.

[7] Dmitrii V. Zagorodnov, Lorenzo Alvisi, Keith Marzullo, and Thomas C. Bressoud. Engineering fault-tolerant tcp/ip services using ft-tcp. In *International Conference on Dependable Systems and Networks*, San Francisco, CA, USA., June 2003.