Implementing Highly-Available WWW Servers based on Passive Object Replication

Roberto BALDONI, Simona BONAMONETA, Carlo MARCHETTI
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, Roma, Italy
{baldoni,bonamone,marchet}@dis.uniroma1.it

Abstract

In this paper we investigate issues related to building highly-available World Wide Web (WWW) servers on workstation clusters. We present a novel architecture that includes a dynamic Domain Name System (DNS) and a WWW server based on passive object replication. This architecture allows to reduce the service down-time of a WWW server without impacting on the Hyper Text Transfer Protocol (HTTP) protocol (and thus on the WWW client software). We implement this architecture in our department and we show some experimental results that analyze, given a batch of HTTP requests, the average response time to a request and the average time of WWW service unavailability due to object crashes.

1. Introduction

World Wide Web (WWW) access has becoming widely popular in the recent years with the consequent exponential increasing of client connections. This effect gave rise to the study and the implementation of scalable and high-performance WWW servers. These servers are usually built on a multi-workstation platform and their aim is to split the (Hyper Text Transfer Protocol) HTTP requests among the workstations in order to share, in a fair way, the load and keep as small as possible the average response time to a client request ([2, 9, 13]). The redundancy of WWW servers provides also some degree of fault tolerance with respect to simple *mirrored* services.

Distributed WWW servers as the ones considered above do not handle a replicated state. i.e., each workstation handles a set of HTTP requests and no information about this handling is given to the other components of the WWW architecture. Ensuring fair load sharing without handling a replicated state among the workstations providing the service is not sufficient to design some key distributed WWW services like the ones, for example, including electronic commerce. As an example, from the viewpoint of the businesses, such WWW servers have to (i) reduce the service down-time as small as possible (i.e., High-Availability), (ii) increase the fault-tolerance of the service as much as possible (i.e., the maximum number of consecutive failures that stop definitely the service and require human actions to restart the service) and (iii) ensure that each workstation providing the service "sees" a *consistent* value for its own replicated state.

This paper focuses on the implementation of highly-available distributed WWW servers over the Internet based on passive object replication. In *passive replication* [11], a given object y is called primary and it is responsible for handling client requests and for maintaining the value of other y's replicas on the other sites consistent with its own value. Once the primary fails, other replicas must elect a new primary. Compared with other replication techniques (e.g. active replication), passive replication seems to be very attactive for the WWW environment as (i) it does not need a client to multicast its invocation to every replica (this is actually not allowed by the HTTP protocol which employs only one-to-one communication between client and server) and (ii) it allows non-deterministic computations.

From a logical point of view, the WWW service is provided by a server realized by cooperating processes residing on the same LAN and handling a set of replicated objects. In order to meet previous third point, our implementation ensures updates of objects satisfying the linearizability correctness criterion for replicated objects [12]. Intuitively, linearizability ensures that each non-crashed replica "sees" the same ordered set of invocations.

From the point of view of the WWW architecture, the fact that the primary can change over the time imposes some modifications. More specifically, the DNS must be aware of the primary address currently providing the service.

This paper is structured in four section. Section 2 briefly describes the current WWW architecture, Section 3 introduces the proposed WWW architecture including the DNS supporting dynamic updates and the distributed WWW server based on passive object replication. Section 4 describes how a prototype of the architecture has been implemented in our department and some experimental results concerning the average service down-time and the average response time to a client request. Section 5 concludes this paper. Due to lack of space implementation details are omitted. Readers can refer to [3] for an exhaustive description of the implementation.

2. Background: The World Wide Web

The WWW architecture can be roughly split into three interacting entities ([1]). The client, running a WWW browser (e.g. NETSCAPE, Explorer, etc.), the Domain Name System (DNS) and the WWW server. Note that while

[1] This is not an exhaustive description of the WWW architecture: many details (for example the caching system) are skipped for simplicity. There are many books describing such an architecture in any detail.

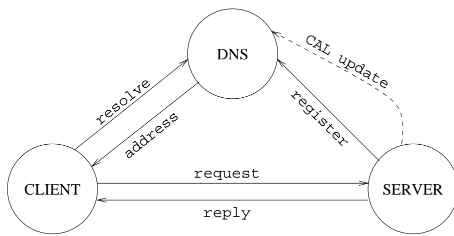

Figure 1. The proposed WWW Interaction

clients are single entities, servers and DNS can be realized by sets of entities. In particular, a server can be implemented by one or more processes and a DNS is structured as a distributed hierarchical database providing the mapping between physical IP addresses and logical names of the services (URL). Hence, DNS is, basically, a database partitioned among its entities and each entity is responsible (*i.e., it is authoritative*) for a portion of the DNS database.

A client wishing to access to a WWW service, in the simplest scenario, it first queries the DNS (`resolve` message in Figure 1) to have back the server IP address (`address` message in Figure 1), then contacts the server by establishing a TCP connection (HTTP request) on that address (`request` message in Figure 1). Finally, the server sends back the required information (HTTP reply) to the client (`reply` message in Figure 1).

The previous protocol cannot work properly if a service has not been registered in its authoritative DNS entity (i.e., the entry corresponding to the pair ⟨service, physical address⟩ is not in the DNS global database). Thus, an initial off-line registration of the service is needed. This communication could be realized by an email exchange or by a telephone call between the DNS system manager and the WWW service system manager.

3. The Proposed Architecture

3.1. Dynamic DNS Updates

Recent discussions about the DNS architecture have pointed out the need of dynamically updating the DNS table [14]. Dynamic updates allow some certified *name-servers* ([2]) to automatically modify the DNS database.

Our idea is to allow *WWW servers* (i.e., workstations which are not name-servers) belonging to a *Certified Address List* (CAL) to send registration messages to their authoritative DNS (e.g., upon the occurrence of a primary crash). Upon the receipt of a certified registration message, name-servers execute an automatic update of the pair ⟨service, physical address⟩ in their database (message `register` in Figure 1). In such a case a first registration is still necessary (`CAL update` message in Figure 1) in order to define which processes have to be included in the CAL of the their authoritative DNS.

The pseudo-code version of the name-server is given in Figure 2. It waits for the arrival of either a `resolve` (line 4) or a `register` (line 9) message ([3]). If a `resolve` message is received, then the RESOLVE_NAME function is invoked (line 6). RESOLVE_NAME is a function that performs a name resolution as described in [1] ([4]). Upon the completion of the name resolution, the required physical address is returned by RESOLVE_NAME function and sent back to the sender (line 7).

When a `register` message is received, it is checked if the sender process is authorized to dynamically update the DNS database (line 10) i.e., its physical address is contained in the local (authoritative) CAL for the specified service. In the affirmative, the UPDATE procedure is invoked. This procedure simply updates the local authoritative table of the DNS database by substituting the previous physical address with the new one. Otherwise, no update is performed at all.

```
procedure NAME_SERVER()
begin                                                                        (1)
  do forever                                                                 (2)
    select                                                                   (3)
      when receive (resolve, service_name, sender_add) from any ;            (4)
        begin                                                                (5)
          phy_add:=RESOLVE_NAME(service_name);                               (6)
          send (address, phy_add) to sender_add;                             (7)
        end                                                                  (8)
      when receive (register, ⟨service_name, primary_add⟩), sender_add)
                                                              from any ;     (9)
        if sender_add ∈ Certified Addresses List
        then UPDATE_TABLE(⟨service_name, primary_add⟩);                     (10)
    endselect                                                                (11)
  od                                                                         (12)
end                                                                          (13)
```

Figure 2. Name-Sever pseudo-code

3.2. WWW Server based on Passive Object Replication

The Model. The system consists of a set of $n > 1$ server processes, a centralized DNS ([5]), a non-empty set of client processes and a non-empty set of objects.

Each server process p_i is augmented with a failure detector process fd_i [5], has access to a local hardware clock perfectly synchronized with respect to real time ([6]) and handles a replica of each object. Note that a server process, its failure detector and its replicas run on the same workstation. A server is able to offer to the clients a set of services. Each service is modeled as a remote method invocation of an object.

Each pair of entities (both processes and DNS) in the system exchange messages over reliable and FIFO communication links. For each link connecting two server processes, it does exist a known time-constant δ such that if processes p_i and p_j are not crashed, then a message sent from p_i to p_j at time t will be received by p_j within $t + \delta$ [7]. This is a reasonable assumption if server processes are connected over a local area network (see Section 4).

Concerning failures, we assume the DNS does not crash and other entities follow a *crash failure* behavior. In particular, we assume that if an entity (i.e., server process, failure detector process, object), residing on a given workstation, fails, all the entities residing on the same workstation fail as well.

---

[2] A name-server is the process managing a portion of the DNS database. In the literature "name-server" and "local DNS" are usually used as synonymous.

[3] Note that manual update commands can be transformed in local `register` messages.

[4] Intuitively, RESOLVE_NAME is a function such that, if the local authoritative DNS contains the required address, then it returns this address, otherwise it sends hierarchically and recursively a `resolve` message to another DNS and waits for a reply. When the reply is received, the physical address is returned.

[5] For simplicity and without lost of generality, we will no longer consider the distributed nature of the DNS database in the following.

[6] Bounded drifts with respect to real time are allowed in practical models. e.g., the Cristian-Fetzer "Timed Asynchronous Model" [7].

Failure Detection. To pursue the aim of implementing highly-available servers, a server process needs to detect the failure of another one. In the past it has been argued that this detection should be decoupled by its client processes and implemented as a middleware service [5, 11, 15].

A server process can ask its failure detector for the most updated state ($alive$ or $crashed$) of the other server processes and can take decision according to that knowledge.

It is well known that detecting failures is difficult in asynchronous distributed systems, in particular, it is practically quite hard to decide if a process is slow or actually crashed ([7]). However, if a WWW server is implemented on a local area network, there are some guarantees on timed message delivery [7] and, as shown in [8], a failure detector process can be obtained by a simple protocol of *independent assessment* using multicast of messages.

Each failure detector process multicasts an `I'm alive` message every τ time-units to all the other failure detectors. A non-crashed failure detector fd_i perceives the crash of p_j if the time elapsed since the last receipt of a fd_j `I'm alive` message is greater than $\tau+\delta$. In such a case the fd_i view of the state of fd_j switches from $alive$ to $crashed$. In the model we consider, this implies the workstation hosting fd_j has halted.

From an operational viewpoint, the message traffic can be reduced either by using the broadcast capability of a LAN or by observing that is not necessary that each server independently detects the failure of all the others. In the latter case, *attendance lists* or a *neighbor surveillance* protocols [6] can be used to reduce to n the number of messages sent every τ time-units.

Primary-Backup. As said in the Introduction, in the primary-backup paradigm, one of the replicas (the *primary*) plays a special role. It receives all the requests from clients, computes the reply and updates all non-crashed replica *backups*. Finally, the primary sends the results back to the client. Under the model of Section 3.2, this paradigm can tolerate up to $n-1$ server process crashes before halting the service.

The pseudo-code of a server process p_i is shown in Figure 3. It first checks if it has been designated to be the primary (line 4). If this is not the case, it enters the while loop and acts as a backup (line 5 – 12), otherwise it goes to line 13. In the while loop a backup waits for one of the following two events:

1.If an `update` message is received from the primary, then it updates the object replica it handles (lines (7–8)),

2.if a crash of the primary has been detected by fd_i (FD($primary_number$)=`dead` - line (9)) then it has to designate a new primary calling the ELECTION function (line 10).

The ELECTION function selects an integer $j \in [1,n]$ such that $\forall i \in [1,j-1] \Rightarrow \text{FD}(i) = dead \wedge \text{FD}(j) = alive$ corresponding to the non-crashed server process p_j with the lowest identifier. As we adopt independent assessment as failure detection mechanism, process p_j has to wait $\tau + 2\delta$ time-units before starting to act as a primary in order each process can complete the crash detection and the election procedure.

A primary process goes to execute line 13. This , as its first action, notifies its address to the DNS by means of a `register` message (line 13), and then it starts the primary code consisting of the execution of an infinite loop (lines 14–26). Upon the receipt of a `request` message, it computes the object state update, the object reply, sends the `update` messages to the non-crashed backups, updates its replica state and finally sends back the reply to the client.

The interaction protocol proposed in Figure 3 is *non-blocking*, in the sense that, the primary does not wait for any `update` acknowledgment from the backups. As remarked in [4], this low-cost interaction can be meet only under assumptions of crash failure and FIFO reliable channels. Note that linearizability is ensured only if (i) the up-

```
init name_server := authoritative_DNS_address;                      (1)

procedure SERVER_i(primary_number)                                  (2)
begin                                                               (3)
 while (i ≠ primary_number) do       % Backup code                  (4)
  begin                                                             (5)
   select                                                           (6)
    when receive (update, update_data) from primary_number          (7)
     COMMIT_STATUS(status, update_data);                            (8)
    when FD(primary_number)=dead                                    (9)
     primary_number:=ELECTION();                                   (10)
   endselect                                                       (11)
  end                                                              (12)

 send (register, ⟨service_name, i⟩, i) to name_server;             (13)

 do forever                     % Primary code                     (14)
  begin                                                            (15)
   receive (request, request_data) from client                     (16)
   update_data :=COMPUTE_STATUS_UPDATE(status, request_data);      (17)
   reply_data :=COMPUTE_REPLY(status, request_data);               (18)
   if update_data ≠ null then                                      (19)
   begin                                                           (20)
    for j ∈ {1,...,n} − {i} do                                     (21)
     if FD(j)=alive then send (update, update_data) to j;          (22)
    COMMIT_STATUS(status, update_data);                            (23)
   end                                                             (24)
   send (reply, reply_data) to client;                             (25)
  end                                                              (26)
 od                                                                (27)
end                                                                (28)
```

Figure 3. The server process p_i pseudo-code.

date (line 21) is executed atomically (i.e., all or none of the replicas will be updated) and (ii) the `request` message has a sufficiently rich semantic to allow a just elected primary to discover that a client request has already caused a replica update. The latter scenario can occur if a primary crashes just before the execution of line 24. In such a case, after the expiration of a time-out, the client could resubmit the request to the new primary which, in turn, has to be able to recognize that the corresponding update was already done. Upon the occurrence of the latter scenario the COMPUTE_STATUS_UPDATE function (line (17)) has to return a special $null$ value such that the new primary immediately sends back the reply to the client (see test on line 19).

4. Implementation and Experimental Results

We realized a prototype of the WWW server over the 10Mbit Ethernet of our department connecting seven workstation SUN. The service provided by our server is a simple counter of client connections. This counter is a replicated object. Each time a client connects to the primary, a method “inc” is invoked. The service ensures that (i) for each pair of client requests a, b, if a arrives before b at the primary, then the counter value returned to a will be lower than the one returned to b. (ii) No counter value is skipped. We implemented such a simple service as our aim is to measure

[7]From a theoretical point of view, this difficulty gave rise to the FLP impossibility result [10].

the average service down-time and the percentual growth of the average response time to a client request upon the occurrance of a primary crash. These values are actually independent from the computational cost of processing of a client request.

4.1. Description of the Experiments

Experiments were carried out injecting failures into the primary. In particular, we vary the number of primary failures f from 0 to 6 and τ (see Section 3.2) from 250 to 1000 msec (by steps of 250 msec.). We measured the average response time of a client request (ART) and the average service down-time (ASDT). i.e., the time elapsed between the crash of a primary and the notification to the name-server of the new primary address by means of the `register` message. To reduce the impact of the variance of the network delay, DNS and HTTP clients were running on the same LAN of the server, where we experienced a value for δ of 70 msec ([8]).

Each point in the plots (Figure 4) has been computed as follows: for each pair $\langle f, \tau \rangle$, we submitted a batch of 1000 HTTP requests to the server and evaluated ART and ASDT. This measurement has been repeated 20 times and, then, the final values of ART and ASDT has been estimated.

4.2. Experimental Results

Average response time. In order to point out the differences between ART values in different scenarios, Figure 4.a plots ART(%) as a function of f for different values of τ, where ART(%) represents the ratio between the ART for a given pair $\langle f, \tau \rangle$ and the ART value in the absence of failures (which is actually independent from τ). As expected,

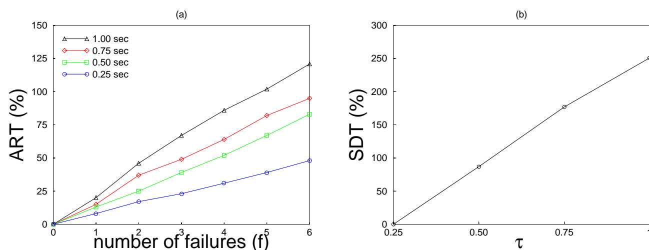

Figure 4. (a) ART(%) as a function of f by varying τ from 0.25 sec. to 1 sec. (b) ASDT(%) as a function of τ

the growth of ART(%) is linear with the number of failures. i.e., k failures lead to k elections and, thus, to increase the average response time of a factor proportional to $k\tau$. Moreover, augmenting τ, ART(%) increases because more time is needed to detect a primary failure and to elect a new primary.

Average service down-time. The last observation becomes more clear looking at Figure 4.b showing ASDT(%) plotted as a function of τ, where ASDT(%) represents the ratio between the ASDT value for a given τ and the ASDT value for τ equal to 0.25 sec. To have a quantitative idea of ASDT we experienced a value of 0.572 sec. for $\tau = 0.25$ sec.

[8] This is the maximum value we measured during several days (including high-traffic hours) on the LAN of our department while experiments were running.

Previous plots can lead system designers to quantify and to reduce as much as possible the service down-time by setting a suitable value for τ, having measured δ provided by their local area networks.

5. Conclusion

A great challenge for companies is how to improve the reliability and the availability of their WWW servers exploiting their internal hardware, software and network resources. In this paper we have described an implementation of an highly-available WWW server based on passive object replication on workstation cluster. This implementation pointed out the necessity of dynamic updates of the DNS database in order to achieve high availability. We presented a novel WWW architecture which allows a certified list of processes to dynamically update the DNS. This architecture reduces the service down-time of a WWW server compared to servers that do not adopt software replication and does not impact on the software of the WWW client being also compatible with the last version of name-server software [1, 14].

References

[1] Albitz P.,Liu C., "Dns and Bind" 3rd Edition, *O'Reilly*,1998.

[2] Andresen D., Ibarra O.H., Holmedahal V., Yang T., "Towards a Scalable Distributed WWW Server on Multicomputers", *In Proc. of International Conference on Parallel Processing*, Honolulu, pp. 850-856, 1996.

[3] Baldoni R., Bonamoneta S., Marchetti C., "Implementing Highly–Available WWW Servers based on Passive Object Replication", *Dipartimento di Informatica e Sistemistica, Università "La Sapienza" di Roma, Tech. Rep. 14-98*, December 1998.

[4] Budhiraja N.,Marzullo K.,Schneider F.B.,Toueg S., "The Primary-Backup Approach", *in Sape Mullender, Editor, Distributed Systems*, ACM Press, 1993.

[5] Chandra T.D., Hadzilacos V., and Toueg S., "The Weakest Failure Detector for Solving Consensus", *Journal of the ACM*, 43, 4, pp. 685–772, 1996.

[6] Cristian F., "Reaching Agreement on Processor-group Membership in Synchronous Distribuited Systems", *Distribuited Computing*, 4, pp. 175-187,1991.

[7] Cristian F., Fetzer C, "The Timed Asynchronous System Model", *In Proc. of the 28th Annual International Symposium on Fault-Tolerant Computing*, Munich, Germany, 1998.

[8] Cristian F., Frank Schmuck "Agreeing on Processor Group Membership in Timed Asynchronous Distribuited Systems", *University of California San Diego, Technical Report CSE95-428,1995*.

[9] Dias D.M., Kish W., Mukherjee R., Tewari R., "A Scalable and Highly Available Web Server", *In Proc. 41st IEEE International Computer Society Conference (COMPCOM)*, Feb. pp. 85-92, 1996.

[10] Fisher M. J., Linch N. A., Paterson M.S., "Impossibility of Distributed Consensus in the Presence of one Faulty Process", *Journal of the ACM*, 32, 2, pp. 374–382, 1985.

[11] Guerraoui R., Shiper A., "Fault-Tolerance by Replication in Distribuited Systems", *in Proc. Reliable Software Technologies – Ada-Europe'96*, Springer Verlag,LNCS 1088, 1996.

[12] Herlihy M., Wing J., "Linearizability: a Correctness Condition for Concurrent Objects", *ACM Trans. on Prog. Lang. and Syst.*, 12,3:463-492, 1990.

[13] Katz E.D., Butler M., McGrath R., "A Scalable HTTP Server:The NCSA Prototype", *Computers Networks and ISDN Systems*, 27, pp. 155–164, 1994.

[14] Thomson S., Rekhter Y., Bound J., "Dynamic Updates in the Domain Name System", *Tech. Rep. RFC 2136*, Internet Engineering Task Force, April 1997.

[15] van Renesse R., Minsky Y., Hayden M., "A Gossip-Based Failure Detection Service", *In Proc. of Middleware '98*, England 1998.