A Generic Reference Software Architecture for Load Balancing over Mirrored Web Servers: NaSr Case Study

R. Bashroush, I. Spence, P. Kilpatrick, TJ. Brown
Queen's University Belfast,
18 Malone Road, Belfast BT7 1NN, UK
{r.bashroush, i.spence, p.kilpatrick, tj.brown}@qub.ac.uk

Abstract

With the rapid expansion of the internet and the increasing demand on Web servers, many techniques were developed to overcome the servers' hardware performance limitation. Mirrored Web Servers is one of the techniques used where a number of servers carrying the same "mirrored" set of services are deployed. Client access requests are then distributed over the set of mirrored servers to even up the load. In this paper we present a generic reference software architecture for load balancing over mirrored web servers. The architecture was designed adopting the latest NaSr architectural style [1] and described using the ADLARS [2] architecture description language. With minimal effort, different tailored product architectures can be generated from the reference architecture to serve different network protocols and server operating systems. An example product system is described and a sample Java implementation is presented.

1. Introduction

Along with the rapid expansion of the internet, the information delivery system using the World Wide Web (hereinafter the Web) has also expanded putting high demand on Web servers. As hardware upgrade has the physical technological limitations, different alternative techniques were considered to enhance web servers' performance.

Mirrored Web Servers is a widely used technology with web services that experience high volume client requests. With mirrored web servers, different machines, with possibly different speeds and memory resources, can be deployed instead of a single server where all carry the same set of services. The mirrored servers can be distributed geographically or situated on the same network. Client access requests are then distributed over the set of clusters to even up the load.

There are several techniques for leading clients to the most efficient server [3][4] and [5][6] according to the constantly changing server and network conditions.

In this paper, we design a generic reference architecture for load balancing over mirrored web servers situated in the same LAN with centralized access via an access serve. We then show an example system tailored for a given specific network and server configuration along with the Java implementation.

Section 2 takes us through the different steps leading to the design of the reference architecture of the load balancer. Section 3 gives an example system generated from the reference architecture designed in 2 and the corresponding Java implementation. Conclusion and future work are finally highlighted in section 4.

2. Designing the reference architecture

The reference architecture is designed adopting the NaSr [1] architectural style (Network Architectural Style for Real-time systems) and described using ADLARS[1] [2], an Architecture Description Language for Real-time Software families.

ADLARS is an architecture description language developed within our research group targeting real-time software product families. NaSr is a network architectural style that was also developed within our research group targeting real-time heterogeneous and distributed systems. Having multi-server environment with heterogeneous operating systems makes from NaSr the perfect style to be used. Dealing with the system as a family of architectures where each architecture is a specific instantiation of the reference architecture for a specific network/server configuration makes from ADLARS a good architecture description language.

[1] ADLARS was developed within our research group first in 1999 as part of the Jigsaw project funded by Nortel Networks ®.

The design stage of any software system starts by capturing the stakeholders/system requirements. There are two main techniques for capturing requirements: *Requirements Engineering* [7] and *Feature Modeling* [9][10]. Requirements engineering is usually adopted by the software engineering research society and the Feature Modeling technique is usually adopted by the Product Line Engineering research society. As our domain of interest is a family of applications rather than a single system, and as we are using ADLARS, which is designed for software product families, we followed the Feature Modeling technique for capturing the system requirements.

The second step after feature modeling is designing the system structure and components. For the structure, we adopted the NaSr style due to the similarity of the solutions provided by this architectural style and our domain of concern (concurrent, real-time, heterogeneous). The design of the system components (and sub-components) is carried out over three phases. The overall process is summarized below and details are given in the following two sections (2.1, 2.2).

Step 1 (feature modeling):

- Phase 0: Designing the feature model of the system.

Step 2 (system design):

- Phase 1: Designing the ADLARS Tasks and the Event Categories (system events). This is a recursive procedure that would require changes to the feature model and the Tasks recursively. Different small testing scenarios might be used to increase confidence in the basic correctness of the task in development.
- Phase 2: Designing the Components. This is a recursive procedure that might require changes to the existing Tasks or feature model (e.g. if you find that two different Components that you put in the same Task require two separate threads of execution, this would require a restructuring of the design). This may also require changes to the above layers.
- Phase 3: Designing the Sub-Components (if needed). This as well might impose changes to the above layers (Components, Tasks and Feature model).

2.1. Feature modeling

In phase 0 we use a feature modeling strategy which is similar to FORM [10] an extension to FODA (Feature Oriented Domain Analysis) [9] to capture the system requirements.

In the feature model tree of our system (not shown due to space limitation) shows that the load balancer should be capable of *reading system/server status* (info about memory, CPU, TCP connections, etc.) and *reporting* this information, every *fixed time interval* or on *status change*, to the access server that keeps a record of each mirror available. The access server should be capable of computing *a load index* out of the status information received from each mirror and sort the list of mirrors in a descending order of efficiency, where the most efficient mirror would be at the top of the list ready for handling the next client request.

2.2. System design

After going recursively through phases 1, 2 and 3, the final architecture of the load balancer consisted of four components (or Tasks as called in ADLARS) in addition to the two default components, the Connection Manager CM and the Service Translation Center STC. The components are as follows (Figure 1):

- *webServerMirror*: reads the status of the mirrored server it is running on (CPU utilization, memory usage, etc.) and reports it to the access server.
- *newMirrorInit*: listens to any newly added mirror to the system and creates a new webServerListener to handle its communication.
- *webServerListener*: listens to the status report sent from a specific webServerMirror it is assigned to and updates the sortedMirrorList with the latest information.
- *sortedMirrorList*: contains a list of all available mirrors sorted with the most efficient server first.

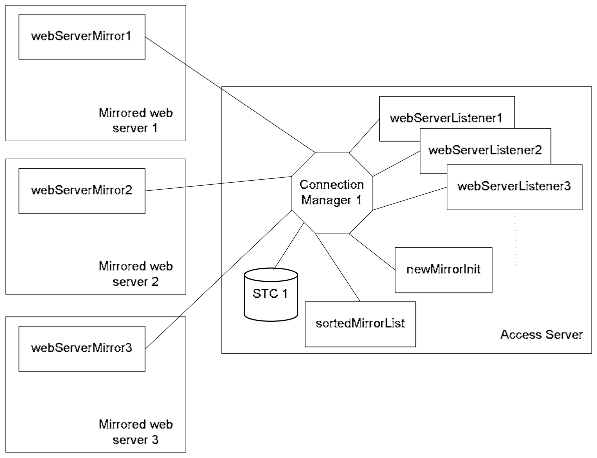

Figure 1. The load balancer architecture

To better understand the design, let us look at the different possible scenarios:

- *New mirror added:* when a new mirror is added to the network
- *Normal operation:* when the available mirrors update the access server with their status
- *Mirror down:* when a mirror becomes unavailable and should receive no more requests (due to overload for example).
- *Mirror recovery:* when a mirror that was down for some reason (overload, power fault, system restart, etc.) is back and operational.

In the first scenario, *New mirror added*, the newly added mirror (represented by *webServerMirror1* lets say, which is an instance of *webServerMirror*) sends an initialization message carrying its machine specifications (CPU speed, memory size, etc.) to the *newMirrorInit* component. The *newMirrorInit* component creates a new instance of the *webServerListener* component, *webServerListener1* for example, to handle the communication of *webServerMirror1*. It also forwards the initialization message of *webServerMirror1* to *webServerListener1* and modifies the STC to forward all incoming messages from *webServerMirror1* to *webServerListener1*. *webServerListener1* would then create a record for *webServerMirror1* in the *sortedMirrorList* component with a load index computed from the information given in the initialization message and according to a given rule (depending on the rule chosen in the specific product architecture, see next section for an example).

In the second scenario, *Normal operation*, a *webServerMirror* should update its corresponding *webServerListener* with its status every fixed interval of time or upon request (depending on the protocol used in the specific product architecture deployed). When the status message is received, a load index is recomputed for that specific mirror, and the *sortedMirrorList* resorts its list with the new value. If a status update message is not received within a time-out interval, the mirror is considered down.

In the third scenario, *Mirror down*, a mirror is considered down by setting the appropriate tag in the *sortedMirrorList* so that it is not included within the sorted list of mirrors, and as a result, no further requests can be forwarded to it.

Finally, in the *Mirror recovery* scenario, a mirror that is tagged as down, would recover from whatever reason prevented it from sending its status update message (overload, power failure, etc.) by sending its status message. From the new status message, the corresponding *sortedMirrorList* would re-compute the mirror load index, unset it's down tag in the *sortedMirrorList* and re-sort the list of mirrors including the recovered mirror so that it can start receiving client requests again according to its position in the list.

The following section presents an example system along with its Java implementation.

3. An example architecture and implementation

In this section, we present an example system we generated from the reference architecture described above.

The system was developed to work over Windows NT® and Solaris® mirrored servers connected over a TCP/IP local network.

The formula used to evaluate the load index over a given mirror is shown in the equation below which was taken from [11].

```
Load Index = LINK*K + IO*L + IDLE*M + CPU*N

 LINK: Number of TCP connections
 IO:   Disk load
 IDLE: CPU idle status
 CPU:  Load average
 K-N: constants
 K=0.2, L=1, M=0.3, N=0.5
```

From these requirements specification, we arrived to the following five Java classes divided into two applications: *mirrorThread.java*, that runs on the different mirrored servers and: *mirrorClass.java*, *arrayList.java*, *mirrorInit.java*, and *mirrorListener.java* which run over the access server.

mirrorThrea.java is the program (has the `main` function) that runs over the mirrored servers. It reads the system information by executing the appropriate command (see feature model for details) via the java command `System.execute("");` and then sends this information to the *mirrorListener.java* class running on the access server via a UDP packet.

mirrorInit.java, is the program (contains the `main` function) that runs over the access server. When started, it creates and runs a thread of *arrayList.java* and listens on a known (to the mirrors) UDP port number for newly added mirrors. The *arrayList.java* (extends Thread) contains an array of mirrors each of which is an instance of *mirrorClass.java* class that contains all the details about a specific mirror. It also includes a `synchronized` function that can be executed by *mirrorListener.java* to update the list of mirrors upon the receipt of new status reports.

When an initialization message is sent by an instance of *mirrorThread.java*, *mirrorInit.java* creates a new instance of *mirrorListener.java* (extends Thread) and assigns it the communication with the initializing *mirrorThread.java* instance.

4. Conclusion

In this research we were concerned with the study and application of our latest Software Engineering and Architecture techniques [13][14] in designing and implementing software solutions for real-life systems [12]. We presented a reference software architecture for load balancing over mirrored web servers. The generic architecture produced served as a case study in using the NaSr architectural style [1]. It also helped us refining the style.

The development of reference architectures is not a new idea and is used in Software Product Line Engineering where Software artifacts are created for product families rather than a specific system increasing reuse and decreasing the production cost and marketing time.

The architecture we designed can be used for quickly developing load balancing systems. By specifying desired features in the reference architecture, the tailored product architecture can be automatically generated using the proper tool. This is one of the main advantages of using our architecture description language ADLARS [2]. An example product system that was generated from the reference architecture was also presented with a minimal level of details due to the space limitation.

5. References

[1] R. Bashroush, I. Spence, P. Kilpatrick, and TJ Brown. A Network Architectural Style for Real-time Systems: NaSr. *Proc. of the 4th Working IEEE/IFIP International Conference on Software Architecture WICSA'04*, Oslo, Norway, June 2004.

[2] T.J. Brown, I. Spence, and P. Kilpatrick. ADLARS: A Relational Architecture Description Language for Software Families. *Proceedings of the Fifth International Workshop on Product Family Engineering*, Siena, Italy, 2003.

[3] A. Myers, P. Dinda, and Hui Zhang. Performance characteristics of mirror servers on the Internet. *Proc. of the Conference on Computer Communications (IEEE INFOCOM)*, New York, Mar. 1999.

[4] M. Crovella and R. Carter. Dynamic Server Selection using Bandwidth Probing in Wide-Area Networks. *Proc. of the Conference on Computer Communications (IEEE INFOCOM)*, Kobe, Japan, Apr. 1997.

[5] M. Conti, E. Gregori, F. Panzieri. Load Distribution among Replicated Web Servers: A QoS-Based Approach. *Proceedings of the 2nd ACM Workshop on Internet Server Performance WISP*, Georgia, May 1999.

[6] R.B. Bunt, D.L.Eager, et al. Achieving Load Balance and Effective Caching in Clustered Web Servers. *Proceedings of the fourth International Web Caching Workshop*, San Diego, CA, Mar-Apr 1999.

[7] R. Abbott and D. Moorhead. Software Requirements and Specifications. Journal of Systems and Software, 2 (1981), pp. 297-316.

[8] Mehdi Jazayeri, Alexander Ran, Frank van der Linden: Software Architecture for Product Families: Principles and Practice, Addison Wesley Longman, 2000.

[9] KC Kang, SG Cohen, JA Hess, WE Novak, AS Peterson: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, CMU/SEI-90-TR-21, ESD-90-TR-222, Nov 1990.

[10] KC Kang, S. Kim, J. Lee, and K. Kim: FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. Annals of Software Engineering, Vol. 5, pp. 143-168, 1998.

[11] T Ogino, M Kosaka, Y Hariyama, K Matuda, K Sudo. Study of an Efficient Server Selection Method for Widely Distributed Web Server Networks. *Proceedings of the 10th Annual Internet Society Conference INET2000*, Yokohama, Japan, July 2000.

[12] R. Bashroush, I. Spence, P. Kilpatrick, and T.J. Brown. A Real-time Network Emulator: ADLARS Case Study. *Proceedings of the Third Asia Pacific International Symposium on Information Technology*, pages 610-617, Istanbul, Turkey, Jan 2004.

[13] R. Bashroush, I. Spence, P. Kilpatrick, and TJ Brown. Towards an Automated Evaluation Process for Software Architectures. *Proc. of the IASTED international conference on Software Engineering SE 2004*, Innsbruck, Austria, February 2004.

[14] R. Bashroush, I. Spence, P. Kilpatrick, and TJ Brown. The Contribution of Architecture Description Languages to the Evaluation of Software Architectures. *Proc. of the National IEEE/IEE/ACM PREP 2004 Conference*, Hertfordshire, UK, April 2004.