Java Autonomous Web Applications with Standalone Entities

Patrick Bellot
Telecom Institute, Telecom ParisTech
CNRS LTCI-UMR 5141
Paris, France
bellot@telecom-paristech.fr

Loïc Baud
Telecom Institute, Telecom ParisTech
CNRS LTCI-UMR 5141
Paris, France
baud@telecom-paristech.fr

Abstract— **In [11], we presented the architecture of SAW, a Web server technology that can be efficiently applied to a wide range of Web applications and services. It is a framework for easy programming of Web applications and services. It is not intended to act as a full and open Web server but is application-dedicated. It has been implemented in Java and used by the authors to propose a fully functional Web application for online conference management. After a general description of the server, we give the design patterns concerning entities management used in its conception. The goal is to have efficient and easy-to-program entities in autonomous Web applications written within the SAW framework.**

Keywords: Web application, Web services, design, development, autonomous server, scalability, entities management, design pattern, Java.

I. INTRODUCTION

There are many ways to implement Web applications and services. We can use WAS (Web Application Servers) such as GlassFish, IBM WebSphere, JOnAS, BEA Weblogic, JBoss, etc. WAS are heavy and powerful tools implementing most of Web 2.0 and 3.0 advances. They are well suited for EIS (Enterprise Information Systems) because all EIS applications and services can be embedded in a single framework resulting in a corporate culture. We can also rely on LAMP architectures (Linux, Apache, MySql, PHP) but these Web applications are sometimes difficult to maintain, to modify and to extend. We can also use CMS (Content Management Systems) that allow elegant template-driven designs of Web sites and provide access to many community tools. However, CMS are not intended to build Web sites with significant business functions.

Our proposal, presented in [11], is to build dedicated autonomous Web applications based on our framework without using any external components (except, of course, the file system). SAW is the concatenation of many interesting paradigms, issued from Web Application Servers technology, that make it easier to program real size applications more efficient than equivalent LAMP applications. SAW stands for Stand Alone Web server and is written in Java. Many Web servers are available as Open Source software [2]. They are able to deliver static contents and some implement Servlet and JSP technologies. SAW tries to avoid WAS weight and complexity while providing the well known WAS triplet: Container, Session beans and Entity beans [7]. Session beans implement the business logic. Containers provides an execution context to the Session beans so that the programmer does not have to worry about technical aspects such as handling TLS/SSL, decoding the request, managing the GET and POST variables or managing the users sessions. Entity beans contain the data of the Web application. Because our Container, Session and Entity are not really Beans in the sense of Java EE6, we provide them under the similar name of Binz.

SAW is not intended to be as powerful as a WAS or LAMP architecture. A SAW server is not an open server where authorized users can deposit various contents. It must be dedicated to a single Web application. In [11], we emphasized that one of the reasons of its efficiency is that it is dedicated to a single application allowing automatic memory or file caching of Session binz outputs and Entity binz data. An application is a Java program where Entities binz are the global data and Session binz are entry points, see Figure 1. Nowadays, there is a real need for high performance and scalable web servers [9]. A SAW application is intended to be executed by any reasonable computer and proxied by an Apache server or a Linux Virtual Server (LVS) if it does not have direct access to Internet. Apache servers are able to efficiently proxy many applications and thus can be a front-end for many back-end applications or services distributed on several back office computers, reducing the load of the font-end server. This ensures horizontal scalability with a low economical impact since SAW does not requires very high computing capabilities.

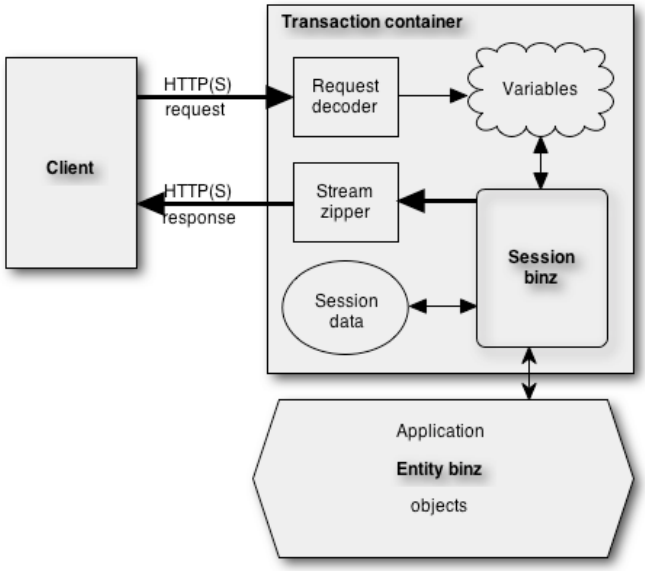

Figure 1 - Architecture of a Web Application

We have used SAW to implement a fully functional and very complete online conference management system [5]. It is operational and has been used for an Asian conference with around 800 users, 150 submitted papers with 3 reviews per paper. The application covers every aspect of a conference or congress including finance.

In this article, we focus on Entities management in SAW autonomous Web applications. We first explain the different patterns we used to manage entities with the notions of Permanent entities, Ephemeral entities and Dispatched entities. Then, we briefly explain how entities are indexed using specialized hash tables. We describe how we implemented Web client agent management of entities using a RESTful approach and how we implement the RESTful approach in a

generic way using Java reflection classes. We show how we use integrated RESTful services in our conference management application. We finally explain how we plan to implement scalability.

II. OVERALL DESCRIPTION

As we said in the introduction, we re-implemented the well-known triplet Container – Session beans and Entity beans. The notion of Beans Container is one of the most interesting WAS innovartions. SAW containers are called Transaction objects. They have been carefully designed and efficiently implemented because they are the core support of all transactions. Transaction objects are Java threads waiting for an incoming request. After processing the request, it returns to its wait state. Thus, there are no dynamic creations of threads. A pool of Transaction objects is created and they are kept ready to handle request. When a request reaches the server, a Transaction object is responsible of its handling. It must manage TLS/SSL encryption if necessary. It decodes the lines of the HTTP request header, the GET parameters provided in the request header path and the POST content if any. If it is proxied by Apache or LVS, there is no need to support SSL connections. According to the header request path, the Transaction object selects the appropriate Session object to answer to the request. It calls the business method of the Session object with itself as a parameter so that the Session object can always access the data and functions of its container. The container is in charge of dynamically loading the Session object class in memory and creating the Session object. Session objects are stored in a hash table where the keys are the request paths. If the Session object class is recompiled, the container will renew the Session instance in memory after reloading the new Session object class. That's a technical reason for the containers to be stateless.

The business method is usually programmed as follows:

- If necessary, the Session object handle method looks at the session user to validate its access to the service. For instance, user's logging and appropriate rights may be required.
- Then, the business method looks at the Transaction variables (GET and POST variables) to validate them. If they are not sound, the Session object can raise an exception asking the Transaction object to log the error, to issue a bad request answer (suspecting a fraud tentative for instance) and then terminates.
- And the method does its business, preparing its response. The Session object has access to the Web application Entity objects containing the application data. As all Session objects have a priori access to all Entity objects, some kind of synchronization must be required depending on the application. The use of a non-coherent object may result in an exception caught by the Transaction object.

The Session objects have access to the Web application data to do its business. These data are global Entity objects.

III. MANAGING ENTITIES

Web application data are usually stored in a relational database and represented by entities. An Entity class matches a table in a relational database and each instance is a row in that table. The mapping of entities objects to the relational database tables is done via hibernation mechanisms such as JPA (Java Persistence API) or another [10]. Access to entities is done via SQL commands and depends on the annotations provided in the class of the entity. For instance, the programmer of the entity class may declare that one attribute of the object corresponds to a primary index field in the corresponding relational database table. Programming this type of entities requires some knowledge and mastering the annotations mechanism but it is rather easy in the powerful IDE (Integrated Development Environment) associated to the Web Application Servers. This approach is interesting and appropriate if there is a very large amount of data to manage.

On the other side, current 2011 computers have high computational capabilities (e.g. 3 GHz and several cores, possibly hyper threaded), big memory cache in processors (up to several MB) and very big RAM memories (several GB is now a common memory size). These huge RAM memories are able to store most of the data of most of the Web applications.

Different entities may not have the same memory requirements. For instance, in an online shop, the entities corresponding to items may require a large amount of memory but a list of admitted customer countries is a very small entity that can be kept in memory for a better efficiency.

We also make a distinction between entities often used and entities rarely used. In SAW, the programmer is free to manage entities using a relational database because she/he has access to the programming power of Java and can use the various databases drivers and hibernation mechanisms. But SAW proposes interesting mechanisms for managing entities: either to keep entities in memory or to store them in disk files. The class **`EntityBinz`** is the main software tool to handle entities. It provides two main generic functions allowing to save entities in disk files and to load entities from disk files:

The class method

`static EntityBinz wakeup(Class eClass);`

reads the entity of class **`eClass`** from the disk file and returns it.

The instance method

`void hibernate();`

saves the calling entity to the disk.

All entities inherit **`EntityBinz`**. The functions use the class name of the entity to compute a name for the file where to save/load the entity. The entity is loaded or saved using Java serialization, thus entities must be serializable. Now, let us consider the three different patterns we used for entities management. The way an entity is managed depends on its usage and its size.

A. Pattern 1 - Permanent entities

In our conference management application, we have an entity that contains users internal identifiers, login name, rights, encrypted passwords and other users data. This entity is not very large (around 5 MB) because we have less than 800 users. This entity must be quickly accessed because it is used at each login on the web site and each operation requiring user rights control. Therefore, it can be kept in memory. Such a Permanent Entity can be declared with the pattern of given in Pattern 1 on next page.

This entity has a unique instance within the application. This instance can be only accessed with the expression **`UsersBinz.getUnique()`**. It must be saved on disk at the server shutdown with a call to **`hibernate()`**. Because the entity is unique through the application, access to its internal data may require synchronization. A power failure is always possible. Thus, hibernation must be ordered at regular time intervals. That's why the entity registers to a backup manager.

B. Pattern 2 – Ephemeral entities

Our conference management application requires an entity containing all countries names. This entity is used to produce the code of a XHTML country selection widget at the start of the server and each time the list of countries is modified. It is also used each time a user modifies its account data to update her/his account data. These are very rare usages in our Web application lifecycle. If needed, the entity is loaded. If the entity is modified, it must be saved using its **`hibernate()`** method. Several instances of the entity, loaded by

```
public class UsersBinz extends EntityBinz
{
     private final static UsersBinz uniqueEntity ;

     static {
            UsersBinz entity = (UsersBinz)wakeUp(UsersBinz.class) ;
            if (entity == null)
                  entity = new UsersBinz() ;
            BackupManager.registerEntity(entity) ;
            OnExitManager.registerEntity(entity) ;
            uniqueEntity = entity ;
     }

     public final UsersBinz getUnique() {
            return uniqueEntity ;
     }

     … // Entity implementation
}
```

Pattern 1: Permanent Entity pattern

different Session binz, may exist at a given time. Thus, if two authorized Session binz modify the entity, the one who hibernates it later wins.

C. *Pattern 3 – Dispatched entities*

Each submitted paper of the conference management application has data such as the paper unique identifier, title, authors, current version, reviewers, reviews and so on. The application must be able to retrieve papers identifiers associated to an author, to retrieve authors associated to a paper, to retrieve reviews associated to a paper, etc. These operations can be done on the numerical identifiers using entities of reasonable memory size (a few MB) such as **`PapersToAuthorsBinz`**, **`PapersToReviewsBinz`**, and so on. However, the data associated to a paper such as the reviews contents are large enough and used so rarely that they can be kept on the disk. But these data are not kept in a single collection entity. Instead, each review will have its own entity saved on disk.

That means that the reviews entity is dispatched in many smaller entities, one for each review. Technically, we keep indexing tables in memory and dispatched data in disk files. That's why we introduced supplementary new wake-up functions with supplementary numeric parameters that are used only for computing more complex file names for the saved entity. And the reviews entities are stored as described in Pattern2. The entity can be loaded only when needed for display or edition. In our application, there will be no conflicting modification of the entity since only the reviewer can modify the review.

IV. INDEXING ENTITIES COLLECTIONS

When we have entities collections such as the collection of all users of our conference management system, indexing these collections is mandatory. Many indexing mechanisms exist, for instance B-Tree was proposed by R. Bayer in 1971 [12]. Using hash table for indexing is known to be a memory waste [6,3].

Because we do not have so much data, we decided to use oversized hash tables. Because Java hash tables only allow objects and no scalar values as keys and because they implement functions that we do not need (automatic size growing, many tests on values and keys), we wrote specialized hash tables classes: one for numerical keys and one for object keys. These hash tables can be used for primary or ordinary indexes. For instance, in our conference management application, we know that the number of users is around 800, probably never exceeding 1024. Thus, we pragmatically chose to index them by their numerical identifier and by their email using hash tables of size 2048. The cost of this table is not significant if compared to the computer memory size. With this scheme, the probability is very high that accessing an indexed user can be done in constant time. And if the number of users increases over our expectation, the only result would be to slow the access and the fix is evident: stop the server and double the size of the two hash tables.

```
public class ReviewBinz extends EntityBinz
{
     public final ReviewBinz getEntity(int id)
     {
            ReviewBinz entity = (ReviewBinz)wakeUp(ReviewBinz.class,id);
            if (entity == null)
                  entity = new ReviewBinz(id) ;
            return entity ;
     }

     … // Entity implementation
}
```

Pattern 2: Dispatched Entity pattern

V. GENERIC RESTFUL SERVICES FOR ENTITIES

Representational State Transfer (REST) is a style of Web services architecture dedicated to resources management [13]. The important concept in REST is the resource that is referenced with a global identifier, for instance a URI in HTTP. To manipulate these resources, client agents and servers communicate via HTTP(S) and exchange representations of resources. The REST server allows the four main CRUD operations:

- (C)reation of a resource,
- (R)ead a resource,
- (U)pdate of a resource and
- (D)eletion of a resource.

The request type identifies the operation: POST for creation, GET for reading, PUT for update and DELETE for deletion. The parameters of the HTTP requests and the answers to the client are represented in a way independent from the resources themselves. Using XML or JSON are usual schemes for representing parameters.

Les us consider the News system of our conference management application. The more recent news is displayed when loading the web site. And the user has the ability to ask for the previous news. The site administrative users can add news or suppress them when required. Let us say that all the conference News are stored on the server side in a memory Permanent Entity of the class named **`data.conference.NewsBinz`** that is a collection of objects of class **`data.conference.News`**. Each News has a random numerical identifier also used for indexing them. It contains the identifiers of the preceding and following News. According to RESTful paradigm, the URI

`http://.../data.conference.NewsBinz/123`

identifies the news with number **`123`**. We chose JSON (JavaScript Object Notation) as the data interchange format [14] because it is lighter and supported by both Java (on the server side) and JavaScript on the client side. Moreover, we don't need the expressive power of XML and we prefer not using the heavy XML libraries.

Our implementation of RESTful entities management system is based on a unique Session binz built using the reflection classes of Java to propose a generic service and listening on a specific port. By extracting the name of the Entity class from the URI, it is able to get the corresponding class object and to recover the Entity object. Then, by extracting the numerical identifier, it is able to recover the corresponding resource and to operate the client request. The reflection classes of Java make this generic RESTful implementation possible. A possibly more efficient solution would be to have a dedicated and automatically generated Session binz for each class of managed Entity. For performance reasons, this is a preferable solution.

VI. RESTFUL CLIENT MANAGEMENT OF ENTITIES

Classically, Web application operates in transactional mode. They propose a page (the View) with a form reflecting the current state of the data (the Model). Local edition is done and the submit button sends the current state of edition to the server that replies with the new state of data.

We found that in some case, it is more interesting to forget about the usual transaction mode. The point is that the View on the client side must always reflect the Model on the server side. This is obtained using hidden light RESTful transactions during the client edition. The transaction is handled by some JavaScript code in the Web client agent. An action of the user on the edition interface results in a RESTful request. If the request fails, the interface must be restored to its previous state. We implemented some specialized pages using this technique when reloading a full page using form submission was really painful.

VII. FUTURE ADVANCES IN SACALABILITY

Due to the increasing usage of Internet B2B and B2C applications, scalability concerns are crucial. Scalability means that we are always able to enhance the server capacity when the traffic increases. A single server that handles all incoming requests does not have the capacity to support high traffic volumes. Increasing the server capacity in terms of processor speed, number of cores or memory size is only a short-term solution [15]. Moreover, this server is a single point of failure and cannot be declared as reliable and available.

Because some operations on entities require a lot of computations and produce peak of memory and CPU usage, for instance getting the list of the users sorted by their names, we now study the feasibility of having external servers dedicated to entities and accessed using Java RMI (Remote Method Invocation). The Web application server would ask entities operations to an entity server. This would reduce the load of the Web application server but it requires more backend communication. Moreover, entities could be shared by different Web applications. A first proof of concept has been implemented.

Our solution consists in externalizing all Entity objects on distinct servers. Thus, we will make the distinction between Session servers, managed by a Linux Virtual Server, and Entity Servers. All communications within this server farm will be transparently done using Remote Method Invocation (RMI). That means that all Entity objects must be remote objects in the sense of RMI. When a Session object requires a data, it asks to the appropriate Entity server that delivers a serialized copy of the data through RMI. If the Session object modifies its copy of the data, it must push it back to the Entity server.

We may also duplicate Entity servers. Two solutions are possible for this redundancy organization. The first solution is to have a complete set of Entity servers associated to each Session server. The second solution is to have a fixed number of occurrences of each Entity server and let the Session servers choose which Entity servers to work with using a round-robin algorithm. When a data modification is pushed on one Entity server, it must be forwarded to all its sibling Entity servers. To avoid a heavy load for the first server, the propagation of the data modification must be propagated in a delegated manner.

We did not find yet a convincing algorithm to properly recover from a general crash of Entities servers. In case of a crash, each Entity server will have its own copy of the data on its local file system at the time of the last backup. But this can lead to an inconsistent situation where the data copies may not agree and where related entities have lost their coherent properties. This is a real problem that needs to be carefully addressed. Moreover, a negative point of our approach is that data are stored in serialized form. That means that we need to write Java program for fixing them.

VIII. CONCLUSIONS

We presented three design patterns for efficient and rational management of entities in Web applications. We also explained how to efficiently manage entities edition using the concept of RESTful service. We claim that most of Web applications do not require databases to support them. Our Entity binz appears as a in-memory database and can be supported by today's hardware.

This approach has proven its viability with an operational web site that was fully reliable and very efficient if compared to the previous Php-MySql server. It has successfully passed the connections peaks resulting from the different conference deadlines.

We have programmed a proof of concept of our approach to scalability. Of course, accessing Entities through Java/RMI is slower than accessing data in memory and this impact the performances of the server. But the server remains truly usable. We had no test for scalability. The Figure 2 shows what we have implemented except the LVS that was replaced by our own load balancer.

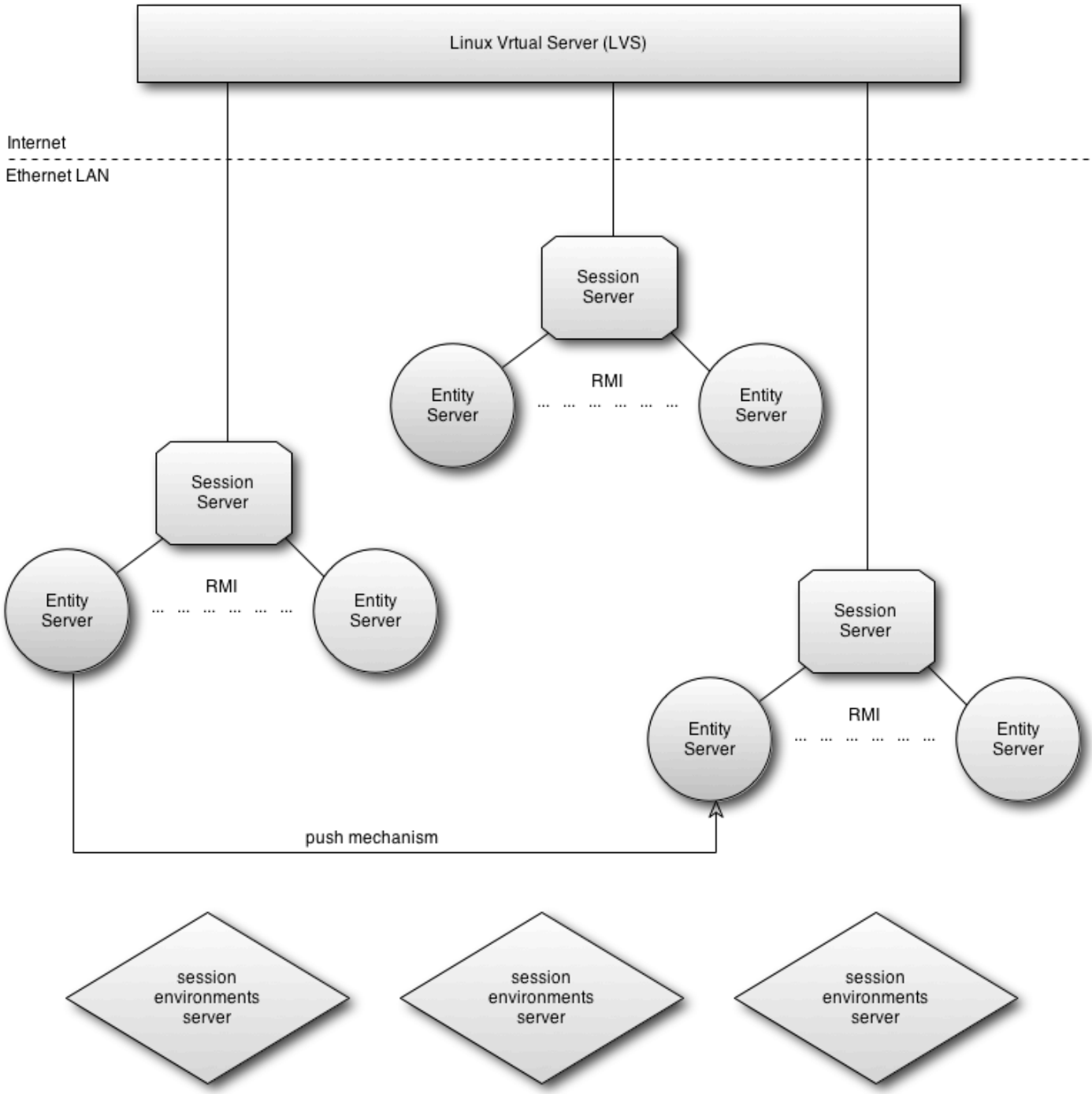

Figure 2 : A scalable architecture

REFERENCES

1. S. A. Walberg. “Tuning LAMP systems’. IBM DeveloperWorks, March 2007.
2. http://java-source.net/open-source/web-servers
3. B. Tindale. “Hash Tables in Java”. The Linux Gazette, n° 57, Sept. 2000.
4. K. Zorgdrager. “Tips and tricks for better Java performance”. IBM Developer Works, May 2001.
5. IEEE-RIVF Conference. http://www.rivf.org
6. H. Garcia-Molina, J. D. Ullman and J. D. Widom. Database Systems: The Complete Book. Prentice Hall, 2nd edition, ISBN-13: 978-0131873254, June 15, 2008.
7. E. Roman, R. P. Sriganesh and G. Brose. “Mastering Enterprise JavaBeans, 3rd edition”, Wiley, 2005.
8. C. Folini. “Apache attacked by a slow loris”. http://lwn.new/Articles/338407/
9. S. Souders. “High performance Web Sites”, O’Reilly Media, Sept. 2007.
10. W. Iverson. “ Hibernate: A J2EE™ Deveoper’s Guide”. Addison-Wesley, 2004.
11. P. Bellot, L. Baud. “A Dedicated Architecture for Efficient Web Server Technology”. The 3rd International Conference on Theories and Applications of Computer Science (ICTACS 2010), Can Tho (Vietnam), Sept. 2010.
12. R. Bayer, E.M. MacCreight. “Organization and Maintenance of Large Ordered Indexes”. Acta Informatica 1, pp. 173-189, 1972.
13. R. Fieldings. “Architectural Styles & the Design of Network-based Software Architecture”. PhD Dissertation. University of California, USA.
14. “JavaScript Programming Language, Standard ECMA-262 3rd Edition”, Dec. 1999.
15. G. Roth. “Server load balancing architectures”. Java World, October 2008.