Analyzing the Impact of Components Replication in
High Available J2EE Clusters

Davide Rossi, Elisa Turrini
Department of Computer Science, University of Bologna
{rossi|turrini}@cs.unibo.it

Abstract

Clustering is a well known technique that allows scalability and fault tolerance in distributed systems. In the J2EE framework, clustering can be used to improve the performance and the availability of an application. In this context, however, great care has to be taken with respect to the replication of stateful components; specifically, stateful components replication can adversely affect the performance of the system and can introduce unexpected modifications in the semantics of the application. In this paper, exploiting our experience in this area, we analyze and discuss these issues, and propose possible solutions. We also describe how component replication is implemented in a number of (open source and commercial) J2EE compliant application servers. The aim of this paper is to provide the insight to better evaluate the software solutions and the related costs/benefits trade-off issues that a software architect has to address when designing a critical, high available, system.

1. Introduction

In the last few years, the success of complex distributed applications, like Web applications or Web Services-based systems, has lead to the need for guarantying non-functional requirements, such performance and reliability.

In the presence of a very large load, clustering is a viable solution to the scalability problem. Clustering enables a group of (typically loosely coupled) servers to operate logically as a single server. Clustered environments allow for horizontal scalability, i.e. the ability to increase the capacity of a system by adding new nodes over which application components can be deployed. This solution is robust (there is no single point of failure), highly available (multiple nodes of the cluster can replicate the same services) and offers high performances (by using load balancing techniques). Several research efforts have addressed clustering technology; for the sake of brevity they are not discuss in this paper. The interested reader can refer to [12,15,17,18,19,20].

It is worth noticing that high availability (HA) in a clustered environment, in the presence of stateful components, requires the replicas (i.e. the versions of a component replicated in multiple nodes of the cluster) to share the same state. Thus, when an invocation changes the state of a replicated component, its replicas have to be synchronized (i.e. they have to reflect the same changes). In this paper, we analyze mechanisms and strategies that can be adopted by a Java 2 Enterprise Edition (J2EE) [1] application server to handle stateful components replication. To this end, we exploit the in-depth knowledge we have gained in this area, as participants of the TAPAS [21] and ADAPT [7] projects. In both projects an existing open source application server (JBoss) has been modified/augmented; in TAPAS by implementing services and mechanisms to meet QoS guarantees, in ADAPT by implementing a framework that allows “pluggable” replication policies.

In this survey, we also discuss how different approaches affect both the performance and the semantics of the application. We also describe how component replication is implemented in open source J2EE application servers or in commercial products for which detailed technical documentation is available.
2. J2EE

J2EE is one of the most popular technologies for the design and development of enterprise applications. The goal of the J2EE platform is to simplify the implementation of the enterprise applications by basing them on modular components, by providing a complete set of services to those components, and by handling many details of application behaviour automatically. Developers are relieved from implementing functionality such as concurrency, transactions, and security management, and can concentrate their efforts in the specific application core-logic.

J2EE is a distributed multi-tiered architecture: as shown in Fig. 1, a typical application is composed by a client tier (usually a Web browser but it can be a specific application as well; in this paper we consider only clients connecting via HTTP), a Web tier (present only when the clients are browsers or software components supporting Web services), a Business tier containing the components that implement the business logic of the application, and an Enterprise Information System (EIS) tier storing the data used by the application. The Web tier and the Business tier contain specific J2EE components: the Servlets and the Enterprise Java Beans (EJBs), respectively, which are deployed in containers. The EIS is usually a relational database connecting to the application server using specific technologies (such as JDBC).

While, as can be seen in Fig. 1, the Web tier and the Business tier are logically split apart, the physical deployment of an application server can be different. In many cases, in fact, it is advisable to run both the Web container and the EJB container in the same host. This allows performing calls from the Servlets to the EJBs as local calls: given the high interaction between these two kinds of components this is one optimisation that can greatly increase the performance of the system. Putting the Web container and the EJB container on the same JVM is usually referred to as co-location.

2.1. State management in J2EE

J2EE components can carry persistent or non-persistent state information. In an application the persistent state is composed by the data of persistent entities (just like an account data in a banking application). Non-persistent state, also called session state or conversational state, consists of data related to entities that exist only during a conversation with a client and that can be discarded once the conversation finishes (in this context a conversation can be a sequence of invocations between remote objects or the navigation of a Web application performed by a human user). The standard example used to better understand the nature of these data is the content of a shopping cart in a Web-based e-commerce application. We want the application to keep these data as long as the user interacts with the application but we do not want to commit these data to a stable storage since we do not need to keep track of what users put in the past into their shopping carts.

In J2EE, the management of persistent data is delegated to the Entity EJBs. They are stateful components with persistent state and are in-memory representation of data committed to some persistent storage hosted at the EIS tier (like rows in a relational database).

Session data can be stored both at the Web tier (using Servlet’s HTTP sessions) and at the Business tier (using Stateful Session EJBs –SFSBs–). Technically speaking, Servlets are not stateful components since they use external shared data structures, the HTTP sessions, to store session state; for the sake of simplicity, however, in this paper we consider Servlets like stateful components.
While both components can be used to store transient data bound to the conversation with a specific client, they really differ with respect to the technologies they use. According to the J2EE design, the granularity and the ability to handle critical loads of work of a component increase moving from the client tier to the EIS tier. This means that session state management via Servlets should be limited to the storage of small amounts of data: when the number of concurrent sessions increases, in fact, the memory needed to store the relative information increases at its turn. EJBs, for example, can count on an automatic mechanism of activation/passivation that moves the data of least recently used SFSBs to secondary storage when the memory available to the container lowers under a given threshold; the same does not apply to HTTP sessions (as a proprietary extension, some J2EE vendor implements a similar behaviour for its application server; but this is not part of the standard).

To complete the comparison between Servlets and SFSB for session state management, we should consider that, in some cases, to store in the Business tier conversational information that are really used only for presentation tasks might introduce the overhead of an inter-tier call (which, depending on the configuration of the application server, can be a remote call). An example of this is an application that allows the users to choose their preferred colour layout for a Web portal. This information is needed by the presentation components (i.e. the Servlets); if we store this information in a SFSB the Servlets have to contact this EJB to retrieve the visual preferences of the user each time a new page is produced.

These considerations help in better understanding that the two options to handle session state information in J2EE are complementary and it is not a good idea to decide to use the one or the other “by design”.

3. Clustering

In this section we analyze how a clustered J2EE architecture deals with load balancing and session state management.

3.1. Load balancing

In order to prevent clients’ HTTP requests to address always the same replica (which would void all benefits of increased reliability and of load balancing) a load balancer component is used to act as a front-end for the application. The load balancer redirects the incoming requests to the replicas of the cluster trying to (as its name implies) balance the work among the nodes. The load balancer is needed by all J2EE application servers that support clustering. It can be hosted in the application server itself but, usually, it is an external component and it is not a standard J2EE component. The load balancer can be implemented either in software or in hardware (hardware load balancers are quite expensive, dedicated network appliances).

The typical configuration of a clustered J2EE environment is depicted in Fig. 3. It is worth noticing that, in order to avoid single points of failure, the load balancer can be coupled with other, passive, load balancers ready to take over the active one in the case of failure (this is usually implemented with dynamic IP address translation or with IP takeover in order to keep transparency with respect to the external clients accessing the application).

3.2. Clustering and state management

In this section we discuss how clustering affects state management considering both persistent and session data.

Clustering interacts adversely on Entity EJBs: all the optimisations that the container can implement when a single host accesses the EIS (like caching database rows in memory) do not apply in the presence of multiple containers. Moreover: we have now the problem of ensuring that concurrent modifications of persistent data during concurrent transaction do not take place. This issue can either be solved by enforcing locks on the data (thus serializing the access to the data: a practice that lowers the concurrency degree of the system and that impacts hugely on the performance) or by forcing the instantiation of only a single Entity bean in the system for a given persistent dataset. Then, all the containers that do not host this specific Entity bean have to perform a remote call to access it. There are alternative solutions to this problem but no one is
contemplated by the J2EE standard. Examples of these solutions are: (i) having read-only copies of an Entity bean or read-mostly copies that cause a failure of the enclosing transaction if two copies change their value during concurrent transactions; (ii) using a shared distributed cache in place of Entity beans for frequently written persistent data. While we are not going to focus on these specific issues, it should be pointed out that they are absolutely relevant when setting up a clustered application server and that overlooking them leads easily to huge bottlenecks in the system.

We now concentrate on session data replication. Both Servlets and SFSBs can be replicated inside a clustered environment. State replication can be implemented either using a shared datastore or by synchronizing the in-memory state of the replicas. With shared data store-based replication, the conversational state is saved in a shared data store like a DBMS or a file system. This approach is quite expensive in term of performance, as the I/O operation on the shared datastore can be time-consuming. Thus we concentrate on in-memory state replication techniques only. By replicating stateful components we can enjoy the benefits of load balancing (since clients’ requests can be redirected to any of the replicas) and we can guarantee high availability (since failed components can be transparently failed over by other replicas). But nothing comes for free: maintaining the conversational state coherence among all replicas introduces an overhead; specifically this approach is only efficient when the number of replicas is small. As show in [14], more the replicas present in a cluster, worst the performance will be. For this reason, with large clusters, it might be useful to replicate components just in a subset of the available nodes. One possible solution is to have a primary-secondary session replication where the session state will only be replicated in one, or maybe two, backup nodes. This way replication overhead is kept low while fault tolerance is guaranteed. An alternative solution is that of splitting the cluster into smaller groups (subclusters or islands); components inside a subcluster are replicated only into nodes of the same subcluster.

In order to guarantee HA and fault tolerance, the subclustering has to support the dynamic reconfiguration, i.e. the ability to (re)distribute dynamically the cluster nodes among the subclusters. With no dynamic reconfiguration the number of nodes in a subcluster can decrease (because of hardware failures, software failures or planned maintenance), and no other nodes are assigned to that subcluster leading to a loss of availability and fault tolerance. Dynamic reconfiguration is a complex issue and in fact no application server we are aware of implements it.

A feature supported by some application server in order to improve the performance of the system even in the presence of a large number of replicas is asynchronous replication. While in a synchronous replication mode the request does not return until the replicated session has been sent over the wire and re-instantiated on all the replicas, in asynchronous replication the request is returned before the update messages have been acknowledged. Of course asynchronous replication offers a smaller degree of reliability: if the client accesses the same session simultaneously using multiple requests (something that can happen even in Web applications, for example in the presence of pages using frames) it is possible that requests go to different nodes with different versions of the session data leading to obvious inconsistency problems.

In the next subsections we discuss specific issues deriving from the state replication both at the Web tier and at the Business tier.

3.2.1. Session state replication at the Web tier

A conversation is a sequence of HTTP invocations from the same client (a Web browser or a SOAP client). As stated previously, at the Web tier the session state of a conversation can be maintained in Servlet’s HTTP sessions.

In a clustered J2EE application server, HTTP requests are dispatched to Servlets by a load balancer. Clustering policies have a big impact on the load balancer: first of all it has to redirect requests originally dispatched to failed nodes to working replicas (i.e. it has to perform fail-over), moreover, in the context of a conversation, it has to redirect all the requests of the user to a node that has the session state related to that session (in the presence of primary/secondary or subclustering replication schemes not all the nodes of the cluster have all the HTTP sessions). The latter problem could be solved by making the load balancer topology aware (that is aware of the replica schema) but, as previously stated, the load balancer might not be part of the application server and could even be a non-configurable hardware appliance. This introduced added complexity to advanced replication mechanisms.

Session affinity, a technique supported by most HTTP load balancers, can come at hand in reducing this complexity. With session affinity the load balancer uses its own balancing policies to redirect requests from a new client to a node but, from that point on, requests
from the same client are redirected always to the same node (this is usually accomplished by using HTTP cookies) unless the node fails. In this case the request is handled as a request from a new client. Primary/secondary replication schemes can take advantage of session affinity: when a request from a new client reaches a Servlet this Servlet becomes a primary, it chooses its secondary node(s) and synchronizes session information with them. If the primary node for that client fails the load balancer redirects the request to another node of the cluster (not necessarily to a secondary one). This node simply broadcasts a message to all other nodes of the cluster asking secondary nodes to synchronize with it and it becomes the new primary.

In the next paragraphs, we are going to present replication component issues that can affect the application semantics.

The first issue we discuss is related to the fact that Servlets are multithreaded components and several concurrent Servlets can access a single HTTP session at the same time (this is quite rare but can happen with multi-frame pages). In this case a Servlet might try to copy a HTTP session to its replicas while another Servlet is changing it, leading to obvious consistency troubles.

The second issue concerns the inherent unreliability of HTTP. While HTTP is based on TCP (at least on the WWW which is based on the Internet), a relatively reliable protocol, it does not support “advanced” delivery guarantees. Scenarios like the following are then possible: a client issues a request that reaches the load balancer. The load balancer dispatches the request to a cluster’s node but it does not get a reply within a timeout. The load balancer then assumes the node has crashed and re-dispatches the request to another node. Of course the load balancer has no clue if the node crashed before or after the synchronization with the replicas. It is then possible that the same request is really processed twice (and if the request is not idempotent this could easily break the application’s semantics [8]). The worst thing is that multiple request processing is completely transparent to the client. Simply adding a unique identifier (like a counter) to clients’ messages would solve the problem (already processed requests would be ignored) but this is not supported by HTTP (something similar would be possible using HTTP cookies but we are not aware of any load balancer/application server that uses this technique).

3.2.2. Session state replication at the Business tier

Enterprise Java Beans are transactional non-multithreading components. Usually a client invocation starts as a HTTP request; the request is dispatched by the load balancer to a Servlet. The Servlet then calls an EJB either with a local java call (in the case of co-location) or by using RMI over IIOP. As soon as the invocation reaches the EJB a transactional context is opened (this is not necessary but it is true with most invocations) and is used both in invocations to other EJBs and in invocations to transactional resource managers. The transactional context is then closed when the first EBJ returns to the Servlet; the Servlet then sends its reply to the client as a HTTP response message.

At the Business tier, the session state can be maintained in the SFSBs. SFSBs’ replica synchronization can take place after each method invocation that changes session information or en bloc at the end of the transaction. In the latter case synchronization can happen before or after the transactional context is closed. If the synchronization takes place after the transaction is closed a crash could cause a missing synchronization after the end of a successful transaction (with obvious unwanted effects related to a new invocation performed for fail-over purposes). On the other side, it is well known that no ACID replication schema can provide high availability, thus moving the synchronization inside the transaction in not a solution in this context. Since no general solution is available for this problem advanced application servers simply move synchronization outside the transaction and perform fail-over in EJB invocations only if all SFSBs’ invoked methods are idempotent (the programmer has to mark each method as such in a configuration file).

4. Analysis of existing J2EE application servers

In this section we analyze how different existing application servers deal with the issues related to session state replication we introduced in Section 3. We included in this survey both open source application servers and commercial, closed source, ones for which in-depth technical documentation is available. Commercial application servers not mentioned here either did not provide adequate technical documentation at the time of this writing, or we have not been able to access it.
4.1. Open source J2EE application servers

Several open source application servers, like JBoss [2], JOnAS [3] and Geronimo [4], implement the J2EE specifications. JBoss and JOnAS have quite similar service oriented architecture. This kind of architecture is flexible as it is possible for a user to configure the application server by adding or removing services. For instance, neither JOnAS nor JBoss have their own Web container, but both can be configured to use an external Web container, like Tomcat [5] or Jetty [6] (both are independent open source project released under the Apache Software license). One of the main differences between JBoss and JOnAS is related to the fact that JBoss provides a more advanced clustering system (for instance it implements Stateful Session Beans replication) with respect to the one found in JOnAS. Both application servers do not provide a load balanced (an experimental load balancer is available for JBoss but it is usable in practice) and suggest [13] the use of mod_jk [11], a module for the Apache Web server.

4.1.1. Session state replication at the Web tier

Replication at the Web tier is handled by the Web container so we now analyze how Tomcat and Jetty deal with it.

Tomcat supports both replication approaches (shared datastore and in-memory) introduced before. If it is plugged in JBoss (Tomcat is its default Web container), it can also exploit the JBossCache service. This service implements a transactional cache that automatically replicates caching data. The cluster behaviour can be customized through a suitable configuration; depending on the various Tomcat version and JBoss integration, several replication options are available. For example, it is possible to decide what triggers a session replication (or when a session is considered dirty) and the replication granularity (the entire session of just the modified fields). Also: both synchronous and asynchronous replication modes are supported. It is also possible to decide to replicate sessions after each HTTP request or to collect sessions for a given period and replicate them en-bloc. This latter option is less reliable but offers better performance. It is also worth noticing that, at the time of this writing, Tomcat performs only an all-to-all replication of session state so, for large clusters, there could be performance problems. To limit this, one can create multiple smaller Tomcat clusters (by using different multicast addresses), but the smaller clusters are not dynamic and in this case it is not possible to use the JBossCache service for replication.

Jetty implements a group communication-based in-memory replication using JGStore. JGStore is an autonomous implementation based on JGroups [8] that is part of Jetty and makes use of no JBoss or JOnAS service. JGStore supports both asynchronous and synchronous replication policies. It is also possible to set hybrid policies, as “GET_FIRST” (the request is returned after at least one node has received the new state) or “GET MAJORITY” (the request is returned after at least the majority of the nodes have received the new state). It is also worth noticing that JGStore supports subclustering. The subclusters configured with JGStore are completely orthogonal to the clusters used by the EJB container. The main shortcoming is that neither Tomcat nor Jetty support dynamic subclustering thus the benefits of load balancing are voided by the overhead introduced by replica synchronization.

While testing the systems we also discovered a “feature” of mod_jk that in some specific situation (e.g. when no replication at all is implemented and the clustering is only used for load balancing purposes) can lead to troubles: once a node of the application server has failed mod_jk “loses” session affinity: the subsequent requests from the client associated to the failed node are load balanced among all the nodes of the cluster even within a single conversation.

4.1.2. Session state replication at the Business tier

JBoss supports SFSBs replication by using HASessionState, a service that is based on group communication. The replication of the state of SFSBs is done at the end of each method invocation. The option to replicate state information only at the end of a transaction is not available in JBoss.

JBoss completely ignores the failure patterns related to multiple invocations of non-idempotent methods of SFSBs, thus a failure can break the semantics (and the data consistency) of the application.

Moreover no advanced clustering techniques are supported. It is indeed possible have more than one cluster and to “emulate” a subclustering setup but the clusters cannot be reconfigured dynamically so this solution does not really make sense.

4.2. Commercial J2EE application servers

At the time of this writing we have been able to identify two commercial J2EE compliant application servers that support clustering and replication of
session state using in-memory group communication-based mechanisms: IBM’s WebSphere [9] and BEA’s WebLogic [10].

Other commercial application servers, like Sun’s SunOne [16] provide HTTP session and SFSBs replication based on a shared datastore only.

### 4.2.1. Session state replication at the Web tier
Both WebSphere and WebLogic implement in-memory HTTP sessions replication based on primary-secondary schemas; WebLogic allows multiple secondary replicas while WebSphere supports just one secondary replica. Both of them offer load balancer support, and implement mechanisms to retrieve a session from another node, in order to guarantee transparent failover. WebSphere can take advantage of a specific load balancer (called sprayer) while WebLogic relies on third part components (the use of a hardware load balancer that supports affine sessions is advised).

### 4.2.2. Session state replication at the Business tier
As before, both WebSphere and WebLogic implement in-memory SFSBs replication based on advanced primary-secondary schemas.

With WebLogic SFSBs methods have to be marked as idempotent in order to activate automatic fail over; we have not being able to find references on how WebSphere handles automatic fail over in the presence of SFSBs; it seems like the “use at your own risk” advice applies to this case.

### 5. Conclusions
In this paper we analyzed mechanisms and strategies that can be adopted by a J2EE application server cluster in order to achieve fault tolerance and high availability. We also pointed out the problems that can arise in this context.

J2EE HA solutions based on component replication should be evaluated with the awareness that specific failure patterns can affect both data consistency and application semantics. It should also be considered that component replication may strongly adversely affect the performance. To avoid this latter drawback, the selected software platform has to support advanced clustering techniques (such as subclustering and primary-secondary). The clustering support offered by existing open source application servers, for example, is by no means of enterprise level when it comes at state replication and fail over. Besides all the limitations we have found in specific implementations, J2EE is possibly the first widely used middleware to adopt in-memory replication techniques (a solution that effectively address strict availability requirements) rather than relying on a shared datastore, an option that is still erroneously perceived as the only real one in order to achieve enterprise-level fault tolerance by large part of the industry.

### 6. References