TheServerSide.com

Under the Hood of J2EE Clustering

Preface

More and more mission-critical and large scale applications are now running on Java 2, Enterprise Edition (J2EE). Those mission-critical applications such as banking and billing ask for more high availability (HA), while those large scale systems such as Google and Yahoo ask for more scalability. The importance of high availability and scalability in today's increasingly inter-connected world can be proved by a well known incident: a 22-hour service outage of eBay in June 1999, caused an interruption of around 2.3 million auctions, and made a 9.2 percent drop in eBay's stock value.

J2EE clustering is a popular technology to provide high available and scalable services with fault tolerance. But due to the lack of support from the J2EE specification, J2EE vendors implement clustering differently, which causes a lot of trouble for J2EE architects and developers. Following questions are common:

- Why are the commercial J2EE Server products with Clustering capabilities so expensive? (10 times compared with no clustering capabilities)
- Why does my application built on stand-alone J2EE server not run in a cluster?
- Why does my application run very slowly in a cluster while much faster in non-clustered environment?
- Why does my cluster application fail to port to other vendors' server?

The best way to understand the limitations and considerations is to study their implementations and uncover the hood of J2EE clustering.

Basic Terminology

It makes sense to understand the different concepts and issues that underlie clustering technology before we discuss the different implementations. I hope this will not only give you the foundation necessary to understand various design issues and concepts in J2EE clustering products, but will also frame the various issues that differentiate clustering implementations and make them easier to understand as well.

Scalability
In some large-scale systems, it is hard to predict the number and behavior of end users. Scalability refers to a system's ability to support fast increasing numbers of users. The intuitive way to scale up the number of concurrent sessions handled by a server is to add resources (memory, CPU or hard disk) to it. Clustering is an alternative way to resolve the scalability issue. It allows a group of servers to share the heavy tasks, and operate as a single server logically.

High Availability
The single-server's solution (add memory and CPU) to scalability is not a robust one because of its single point of failure. Those mission-critical applications such as banking and billing cannot tolerate service outage even for one single minute. It is required that those services are accessible with

reasonable/predictable response times at any time. Clustering is a solution to achieve this kind of high availability by providing redundant servers in the cluster in case one server fails to provide service.

Load balancing
Load balancing is one of the key technologies behind clustering, which is a way to obtain high availability and better performance by dispatching incoming requests to different servers. A load balancer can be anything from a simple Servlet or Plug-in (a Linux box using ipchains to do the work, for example), to expensive hardware with an SSL accelerator embedded in it. In addition to dispatching requests, a load balancer should perform some other important tasks such as "session stickiness" to have a user session live entirely on one server and "health check" (or "heartbeat") to prevent dispatching requests to a failing server. Sometimes the load balancer will participant in the "Failover" process, which will be mentioned later.

Fault Tolerance
Highly available data is not necessarily strictly correct data. In a J2EE cluster, when a server instance fails, the service is still available, because new requests can be handled by other redundant server instances in the cluster. But the requests which are in processing in the failed server when the server is failing may not get the correct data, whereas a fault tolerant service always guarantees strictly correct behavior despite a certain number of faults.

Failover
Failover is another key technology behind clustering to achieve fault tolerance. By choosing another node in the cluster, the process will continue when the original node fails. Failing over to another node can be coded explicitly or performed automatically by the underlying platform which transparently reroutes communication to another server.

Idempotent methods
Pronounced "*i-dim-po-tent",* these are methods that can be called repeatedly with the *same* arguments and achieve the *same* results. These methods shouldn't impact the state of the system and can be called repeatedly without worry of altering the system. For example, "getUsername()" method is an idempotent one, while "deleteFile()" method isn't. Idempotency is an important concept when discussing HTTP Session failover and EJB failover.

What's J2EE Clustering?

A naive question, isn't it? But I still use some words and figures to answer it. Generally, the J2EE clustering technology includes "Load balancing" and "failover".

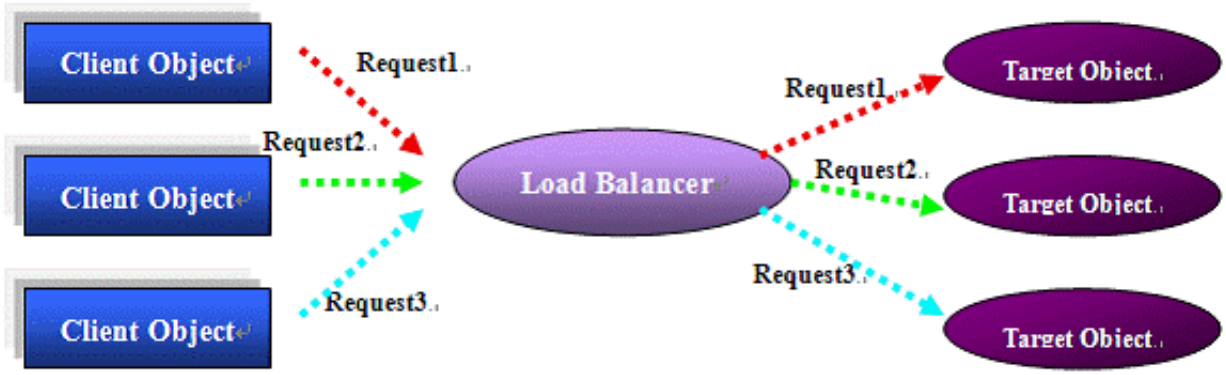


Figure 1: Load balancing

Shown as figure 1, load balancing implies that there are many client objects which make requests to target objects concurrently. A load balancer which sits between the callers and callees can dispatch the requests to the redundant objects which have the same functions as the original one. High availability and high performance can be achieved in this way.

Figure 2: Failover

Shown as figure 2, failover works differently from load balancing. Sometimes, the client object will make successive method requests to a target object. If the target object fails between the requests, the failover system should detect this failure and redirect the later requests to another available object. Fault tolerance can be achieved in this way.

If you want to know more about J2EE clustering, you should ask more such basic questions as “what types of objects can be clustered?” and “where will load balancing and failover happen in my J2EE code?” Those are very good questions to understand the principle of J2EE clustering. In fact, not every object can be clustered, and not everywhere in your J2EE codes, load balancing and failover can happen! Have a look at the following codes:

```
public class A {
……
public void business() {
    B instance1 = new B();
    instance1.method1();
    instance1.method2();
    ……..
}
……..
}
```

```
public class B {
……
public void method1() {
    ……
}

public void method2() {
    ……
}
```

Figure 3: code sample

Will the codes in the "business()" method in "Class A" be load balanced or failed over to another B instance when "instance1" fails? No, I don't think so. For load balancing and failover, there must be an interceptor between the caller and the callee to dispatch or redirect the method calls to the different objects. The objects of Class A and B are running in the same JVM and coupled tightly. It is hard to add some dispatching logic between methods calling.

So, what types of objects can be clustered? – **Only those components that can be deployed in distributed topologies**.

So, where will load balancing and failover happen in my J2EE code? – **Only where you are calling a distributed object's methods.**

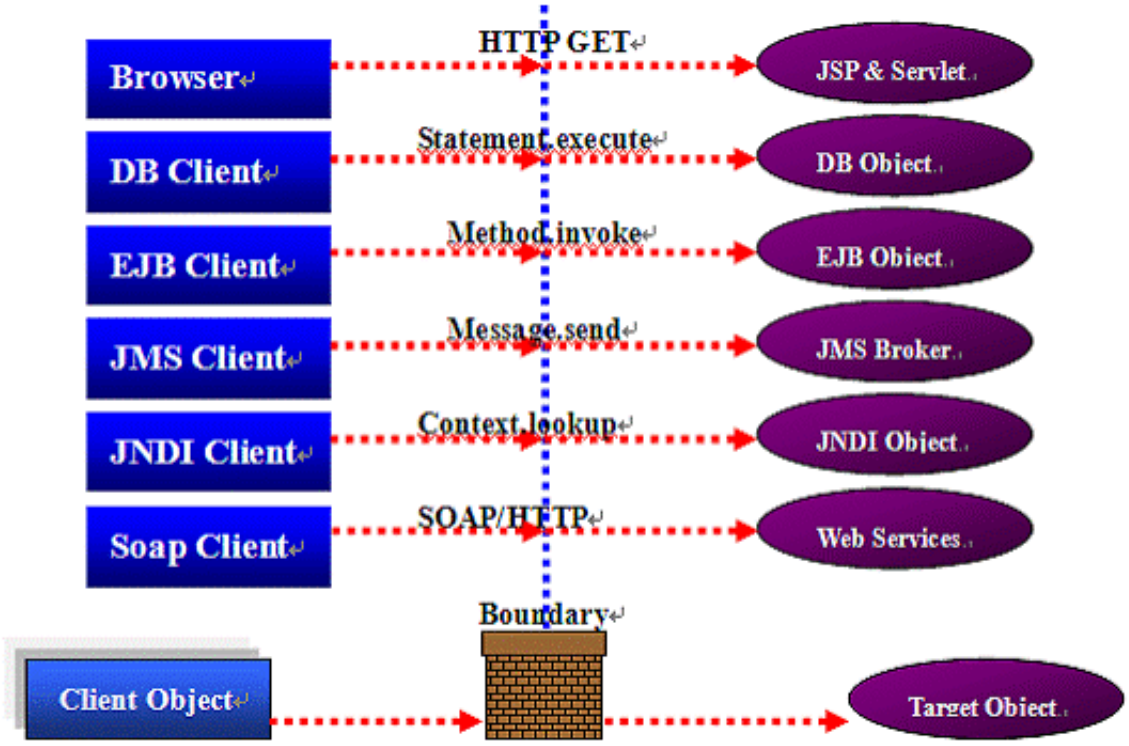


Figure 4: Distributed Objects

In a distributed environment, shown as figure 4, the callers and callees are separated into different runtime containers with obvious boundary. The boundary can be between JVMs, processes or machines.

When the target object is called by client, the function is executed in the target's object's container (that's why it is called distributed). Clients and target objects communicate through standard network protocol. These features give chances for some mechanisms to interfere into the method calling route to achieve the load balancing and failover.

Shown as figure 4, a browser may call a remote JSP object through HTTP protocol. The JSP is executed in a Web server, and the browser doesn't care about the execution, it only wants the results. In such scenario, something can sit between the browser and the Web server to achieve the load balancing and failover functions. In J2EE, distribute techniques include: JSP(Servlet), JDBC, EJB, JNDI, JMS, Web services and others. Load balancing and failover can happen when these distributed methods are called. We will discuss the detailed techniques in the next sections.

Web tier clustering implementation

Clustering in the Web tier is the most important and fundamental function in J2EE clustering. Web clustering technique includes: Web load balancing and HTTPSession failover.

Web Load Balancing

The J2EE vendors achieve Web load balancing in many ways. Basically, a Load balancer intervenes between browsers and Web servers, shown as figure 5.

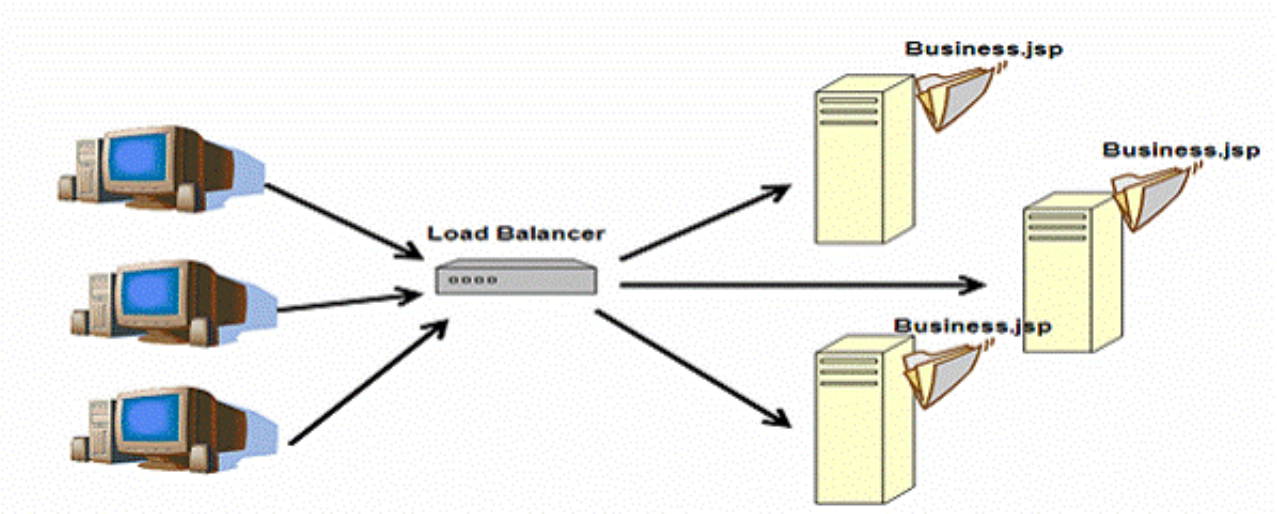


Figure 5: Web Load Balancing

A load balancer could be a hardware product such as F5 Load Balancer, and it also could just be another Web server with Load Balancing Plug-Ins. A simple Linux box with ipchains can also perform load balancing very well. Whatever technique it uses, the load balancer normally has the following features:

- **Implement Load balancing algorithms**
  When client requests comes, the load balancer will decide how to dispatch the requests to the backend server instances. Popular algorithms include Round-Robin, Random and Weight Based. The load balancer tries to achieve equal work load to every server instances, but none of above algorithms can really get ideal equality because they are only based on the number of requests dispatched to a certain server instance. Some sophisticated load balancer implements special algorithm which will detect every server' works load before dispatching the requests to the servers.
- **Health check**
  When some server instance fails, the load balancer should detect this failure and never dispatch requests to it any more. The load balancer also needs to monitor when the failed server comes back, and resume dispatching requests to it.
- **Session stickiness**
  Nearly every Web application has some session state, which might be as simple as remembering whether you are logged in, or the contents of your shopping cart. Because the HTTP protocol is itself stateless, session state needs to be stored somewhere and associated with your browsing session in a way that can be easily retrieved the next time you request a page from the same Web application. When load balancing, it is the best choice to dispatch the request to the same server instance as the last time for a certain browser session. Otherwise, the application may not work properly.

Because the session state is stored in the memory of certain Web server instances, the feature of "session stickiness" is important for load balancing. But, if one of the server instances fails due to some reasons such as power off, all the session state in this server will get lost. The load balancer should detect this failure and won't dispatch any requests to it any more. But those requests whose session state was stored in the failed server will lost all the session information, which will cause errors. That's where session failover comes!

HTTPSession Failover

Almost all popular J2EE vendors implement HTTPSession failover in their cluster products to ensure that

all client requests can be processed properly without losing any session state in case of failure of some server instances. Shown as figure 6 , when a browser visits a stateful Web application(step 1, 2), this application may create a session object in memory to store information for later use; And at the same time, send the browser a HTTPSession ID which can identify this session object uniquely(step 3). The browser stores this ID as a "Cookie", and will send the "cookie" back to Web server when next time it requests a page from the same Web application. In order to support session failover, the session object in the Web server will backup itself somewhere sometime (step 4), to prevent session lost in case of server failures. The load balancer can detect the failure (step 5 ,6), dispatch the sequent requests to another server instance which installed the same application (step 7). Since the session object is backed up somewhere, this new Web server instance can restore the session (step 8) and process the requests properly.

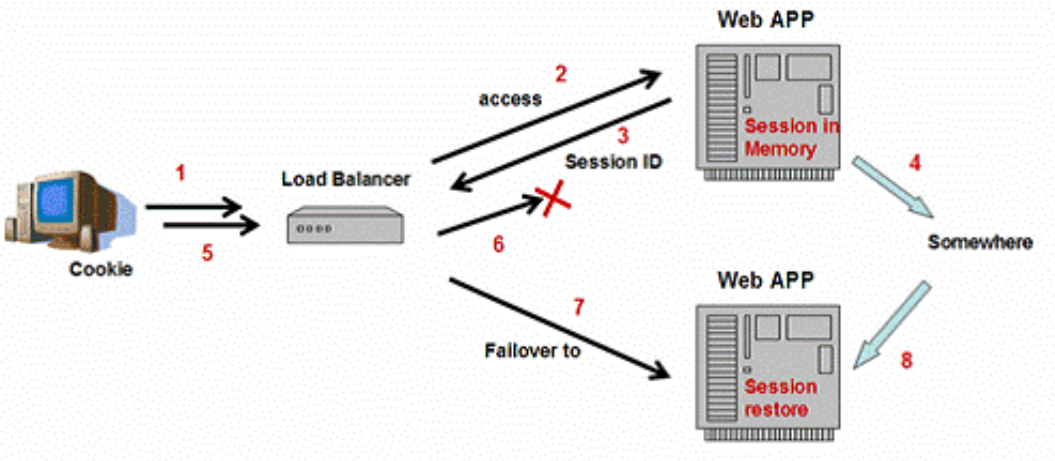


Figure 6: HTTPSession Failover

To realize the above functionality, following issues should be taken into HTTPSession failover implementations

- **Global HTTPSession ID**
  As described above, a HTTPSession ID is used to identify an in-memory session object in certain server instance uniquely. In J2EE, HTTPSession ID depends on JVM instances. Every JVM instance can hold multiple Web applications, each of these applications can hold many HTTPSessions for different users. HTTPSession ID is the key to access the related session object in current JVM instance. In session failover implementations, it is required that different JVM instances should not produce two identical HTTPSession Ids, because when failover happens, sessions in one JVM may be backed up and restored in another. So, a global HTTPSession ID mechanism should be established.
- **How to backup session states**
  How to backup the session states is a key factor to make one J2EE server special and outstanding from the others. Different vendors implement it differently and I will explain this in detail in the next sections.
- **Backup frequency and granularity**
  HTTPSession state backup has performance costs, including CPU cycles, network bandwidth and IO cost of writing to the disk or database. The frequency of backup operations and the granularity of backup objects will impact the performance of the cluster heavily.

Database persistence approach

Almost all popular J2EE cluster products will let you choose to backup your session state to a relational Database through JDBC interface. Shown as figure 7, this approach is simply to let server instances to

serialize the session contents and write to a database at proper time. When failover happens, another available server instance takes the responsibility for the failed server, and restores all session states from the database. Serialization of objects is the key point, which make the in memory session data persistent and transportable. More information about Java Object Serialization, please refer to "http://java.sun.com/j2se/1.5.0/docs/guide/serialization/index.html".

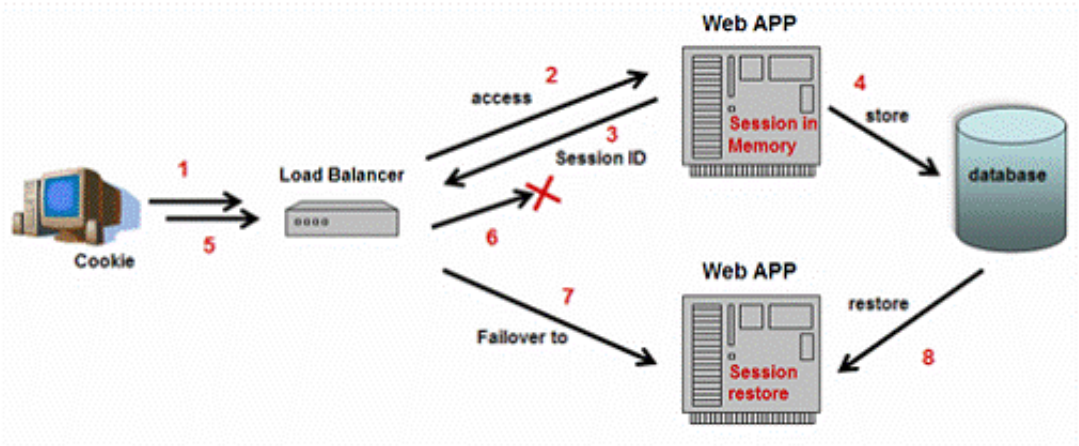


Figure 7: Backup Session State to a Database

As database transactions are expensive, the main drawback of this approach is limited scalability when storing large or numerous objects in Sessions. Most application servers that utilize database session persistence advocate minimal use of HTTPSessions to store objects, but this limits your Web application's architecture and design, especially if you are using HTTPSession to store cached user data.

The database approach has some advantages though.

- It is simple to implement. Separate requests processing from session backup processing make a cluster more manageable and robust.
- Session can fail over to any other host because database is shared.
- Session data can survive the failure of the entire cluster.

Memory replication approach

Due to performance issues, some J2EE Servers (Tomcat, JBoss, Weblogic, and Websphere) provide an alternative implementation: in-memory replication.

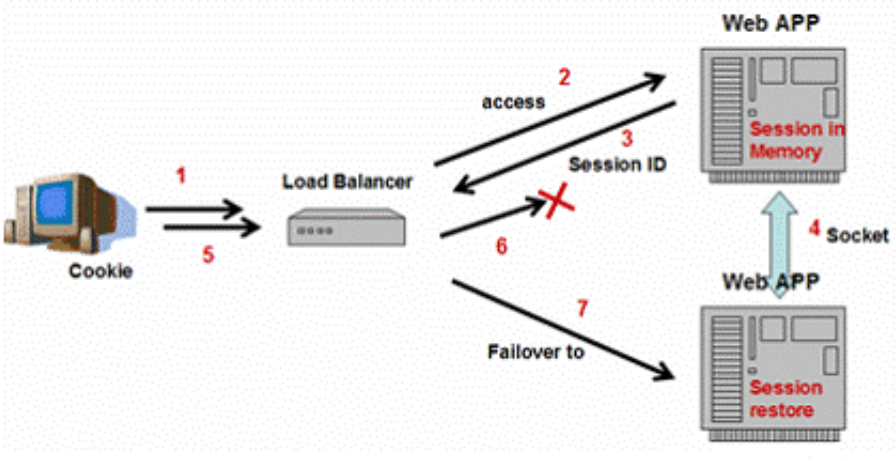


Figure 8: Memory replication for Session State

Memory-based session persistence stores session information in the memory of one or more backup servers instead of Database (shown as figure 8). This approach is very popular due to its high performance. Comparing with database approach, direct network communication between the original server and backup servers is really lightweight. And also note that in this approach, the "restore" phase in Database persistence approach is not needed because after session backup, all session data is already in the backup servers' memory for the coming requests.

"JavaGroups" is currently the communication layer of JBoss and Tomcat clustering. JavaGroups is a toolkit for reliable group communication and management. It provides core features such as " Group membership protocols" and "message multicast", which is very useful in making clustering work. For more information about "JavaGroups", please refer to "http://www.jgroups.org/javagroupsnew/docs/index.html".

Tomcat's approach :Multi-servers replication

Many variations of memory replication exist. The first method is replicating the session data across all of the nodes in the cluster. Tomcat 5 implements memory replication in this way.

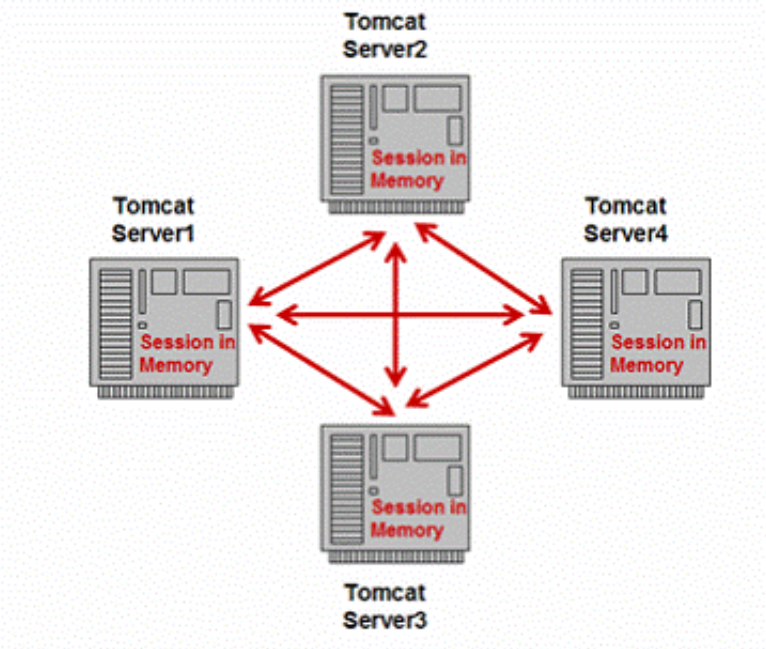


Figure 9: Multi-servers replication

Shown as figure 9, when sessions change in one server instance, it will backup its data to all other servers. When one server instance fails, the load balancer can choose any of other available server instances as its backup. But this approach has some limitations in scalability. If there are too many instances in a cluster, the network communication cost cannot be ignored, it will decrease the performance heavily and network traffic may also be a bottleneck problem.

Weblogic, Jboss and WebSphere's approach-- paired servers replication

For performance and scalability reasons, Weblogic, JBoss and Websphere all provide another way to perform memory replication: each server instance chooses an arbitrary backup instance to store session information in-memory, shown as figure 10.

In this way, every server instance has its own paired backup server instead of all other servers. This approach eliminates the scalability problems when more instances are added to the cluster.

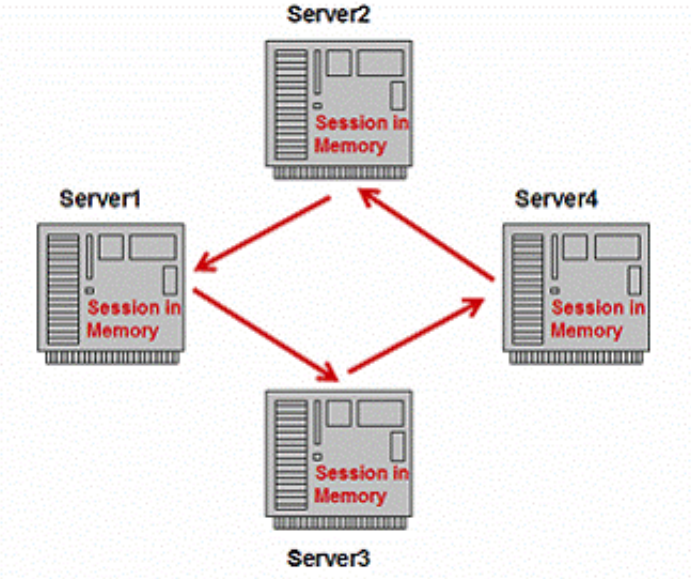


Figure 10: paired servers replication

Although this approach brings an implementation of session failover with high performance and high scalability, it still has following limitations:

- It brings more complexity to load balancer. When one server instance fails, the load balancer must remember which instance is the paired backup server of the failed one. This will diminish the scope of load balancers, and some hardware boxes cannot be used in such requirement.
- In addition to normal request processing, the servers are taking on replication responsibility as well. Per Server instance, request processing capacity is diminished because some of the CPU cycles are now going toward replication duties.
- In normal processing, a lot of memory which is used to store backup sessions wastes in every backup servers when no session failover happens. This can also increase JVM's GC overhead.
- The server instances in a cluster form replication pairs. So if the primary server which the sessions is stuck to fails, the load balancer can send all failover requests to the backup server. The backup server will see doubling in incoming requests after the primary fails and this will cause performance problems in the backup server instance.

To overcome above limitations, variations from different vendors come into being. To overcome the 4 th point above, Weblogic defines the replication pairs for each session instead of each server. When one server instance fails, the sessions in the failed server have dispersed backup servers, and load gets evenly spread after failure.

IBM's Approach -- centralized state server

Websphere has an alternative choice to memory replication: Backup Session information to a centralized state server, shown as figure 11.

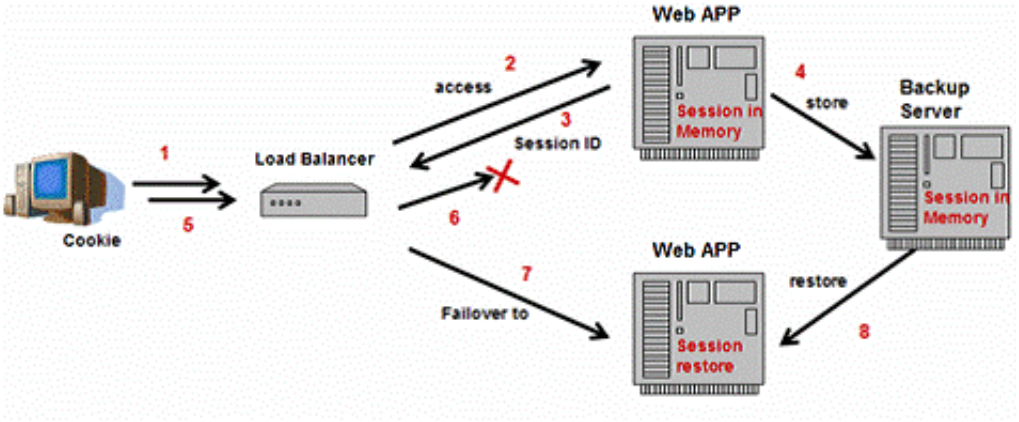


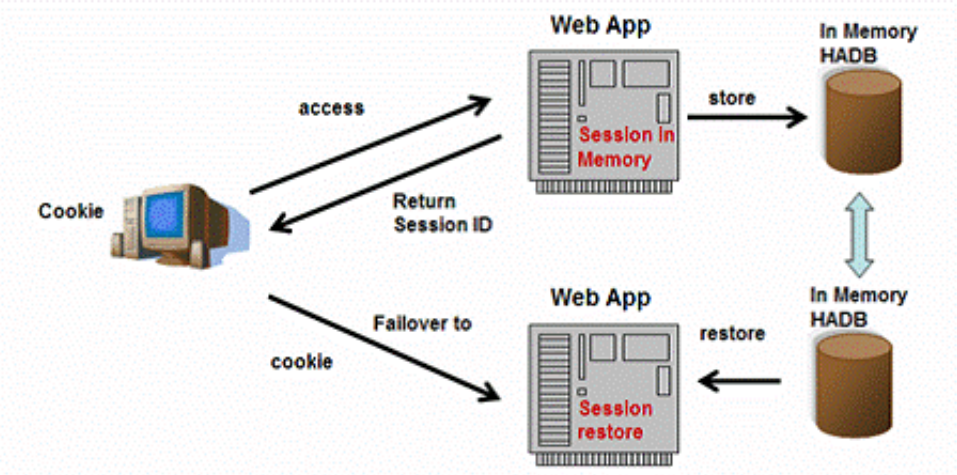
Figure 11: Centralized Server Replication

It's very similar to the database solution. The difference is that a dedicated "Session backup Server" replaces the database server. This approach brings combined advantages from both database and memory-replication solutions:

- Separating requests processing from session backup processing makes the cluster more robust.
- All the session data is backed up to the dedicated servers. No need for server instances to waste memory for backing up session data from other servers.
- Session can fail over to any other instances, since the session backup servers are shared by all nodes in the cluster. So, most of load balancer software and hardware can be the choice for the cluster; and more important, the request loads will spread evenly when one server instance fails.
- Because the socket communication between Application server and session backup server is lightweight comparing to the heavy database connections, it has better performance and is more scalable than the Database solution.

However, due to the "restore" phase to recover the session data for the failed server, its performance cannot be the same as the solution in which memory is replicated directly between pairs. Also, the additional session backup servers add more complexity to administrators, and it is more likely to form the performance bottleneck in the backup sever itself.

Sun's approach – special Database



Sun JES Application Server implements Session Failover differently,shown as figure 12. From the surface, it looks like the same as the database approach, because it uses a relational database as the

Session store and use JDBC to access all the session data. But from the internal, the relational database used by JES, which is called HADB, is optimized for session access specially and stores almost all data in memory. So, you can say it is more close to the approach of centralized state server.

Figure 12: Special Database Replication

Performance issues

Think about this: One Web server may host dozens of Web applications, each of which may be accessed by hundreds of concurrent users, and every user will generate his browser sessions to access certain applications. All the session information needs to be backed up in case of the server failure to restore the sessions in another available server instance. And even worse, the sessions are changing from time to time: when sessions are created or expired, when attributes are added to or remove from the sessions and when attributes are modified; even when no attributes are updated, the last modified time of the session is changing when accessing (to decide when to expire the session). So the performance is the big issue in the session failover solutions. Vendors always give you some tunable parameters to adjust the server's behavior to meet your performance requirement.

When to backup sessions

When client requests are processed, the session data is changing every time. For performance issue, it is not wise to backup sessions in real time. It is really a trade-off to choose the backup frequency. If backup actions take place too frequently, the performance will be impacted sharply; But the longer the interval between two backup actions is, the more session information you will lose if a server failure happens during the interval. In spite of all approaches, including database and memory replication, followings are popular options to decide the backup frequency:

- By Web-methods.
  The session state is stored at the end of each Web request prior to sending a response back to the client. This mode provides the best guarantee that the session state is fully updated in case of failure.
- By Constant interval.
  The session state is stored in the background at the frequency of a constant interval. This mode does not guarantee that session state is fully updated. However, it can provide a significant performance improvement because the state is not stored after each request.

Backup granularity

When backing up sessions, you still have choices to decide how much of the session state is stored. Some common choices among different products are:

- Whole session.
  The entire session state is stored every time. This mode provides the best guarantee that your session data is correctly stored for any distributable Web application. This approach is simple and adopted in both memory replication solution and Database persistence approach by default.
- Modified session.
  The entire session state is stored if it has been modified. A session is considered to have been modified if "HTTPSession.setAttribute()" or "HTTPSession.removeAttribute()" was called. You must guarantee that these methods are called every time an attribute is changed. This is not a J2EE specification requirement, but it is required for this mode to work properly. Modified session backup cuts down the number of sessions to be stored. Those requests which only read attributes from sessions will not trigger the session backup actions, that brings better performance than the

whole session mode.

- Modified attribute.
  Only modified session attributes are stored instead of the whole session. This minimizes the session data to be backed up. This approach will bring the best performance and least network traffic. For this mode to work properly, you must follow a few guidelines. First, Call “setAttribute ()” every time the session state is modified and only the modified object is serialized and backed up. Second, Make sure there are no cross-references between attributes. The object graph under each distinct attribute key is serialized and stored separately. If there is any object cross references between the objects under each separate key, they will not be serialized and deserialized correctly. For example, in a memory replication cluster, shown as figure 13, there are a “school” object and a “student” object in the session and the “school” object has a reference to the “student”. The “school” object is modified at sometime and backup itself to the backup server. After serialization and deserialization, the restore version of “school” object in the backup server will include the whole object graph and contain a “student” object with a reference to it. But the “student” object can be modified separately. When it is modified, only the “student” itself will be backed up. After serialization and deserialization, a “student” object is restored in the backup server’s memory, but at this time, it will lose the connection with “school” object. Although this approach brings the best performance, the above limitation is imposed on Web application's architecture and design, especially if you are using the session to store cached complex user data.

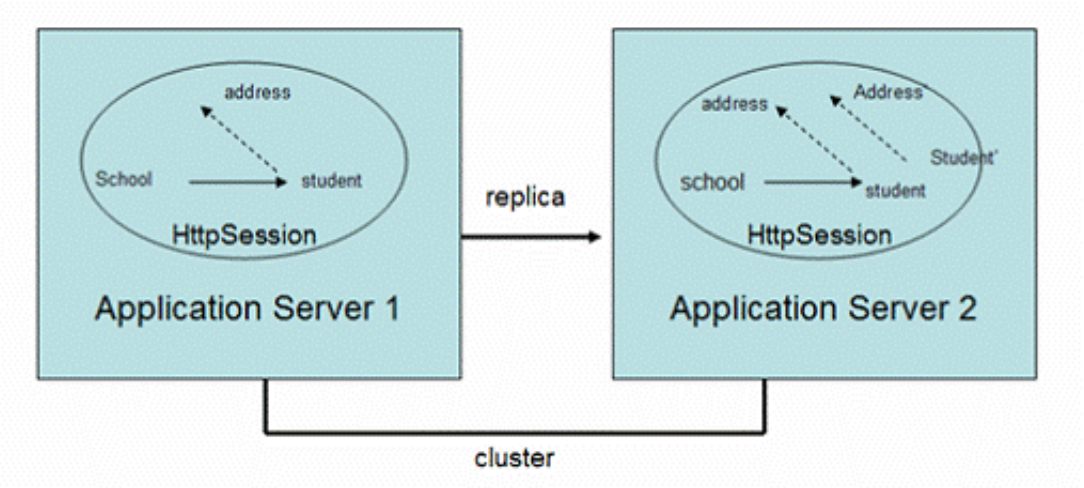


Figure 13: Cross References in Session Replication

Other failover implementations

As I mentioned in the above section, granularity is very important to performance when sessions are backed up. However, current implementations, both database persistence and memory replication, are all using Java object serialization technology to transfer the java objects. This will bring a big footprint, impact system’s performance and also give a lot of limitations on Web application's architecture and design. Some J2EE vendors seek special means to implement web clustering in a lightweight, small footprint mode and provide fine-granularity distributed-object sharing mechanism to improve cluster performance.

JRun with Jini

JRun 4 has built their clustering solution based on Jini technology. Jini was born for distributed computing and it allows you to create a "federation" of devices and software components in a single distributed computing space. Jini provides the distributed system services for look-up, registration, and

leasing which is useful to a clustering environment. Another technology called JavaSpace built on Jini provides features such as object processing, sharing, and migration which are also valuable to a cluster implementation. For more information about Jini and JavaSpace, please refer to "http://java.sun.com/products/jini/2_0index.html".

Tangosol with Distributed Cache

Tangosol Coherence™ provides a distributed data management platform which can be embedded into most of popular J2EE containers to provide clustering environment. Tangosol Coherence™ also provides distributed cache system which can share java objects among different JVM instances effectively. For more information about Tangosol, please refer to "http://www.tangosol.com/".

JNDI clustering implementation

The J2EE specification requires that all J2EE containers should provide an implementation of the JNDI specification . The primary role of JNDI in a J2EE application is to provide the indirection layer so that resources can be found without being much aware of the indirection. This will make J2EE components more reusable.

Having full-featured clustered JNDI is important for a J2EE cluster, as almost any EJB access starts with a lookup of its home interface in the JNDI tree. Vendors implement JNDI clustering differently, depend on their cluster structure.

Shared global JNDI Tree

Both Weblogic and JBoss have a global, shared, cluster-wide JNDI Context that clients can use to lookup and bind objects. Things bound to the global JNDI Context will also be replicated across the cluster through IP multicast so that if a server instance is down, the bound objects will still be available for lookup.

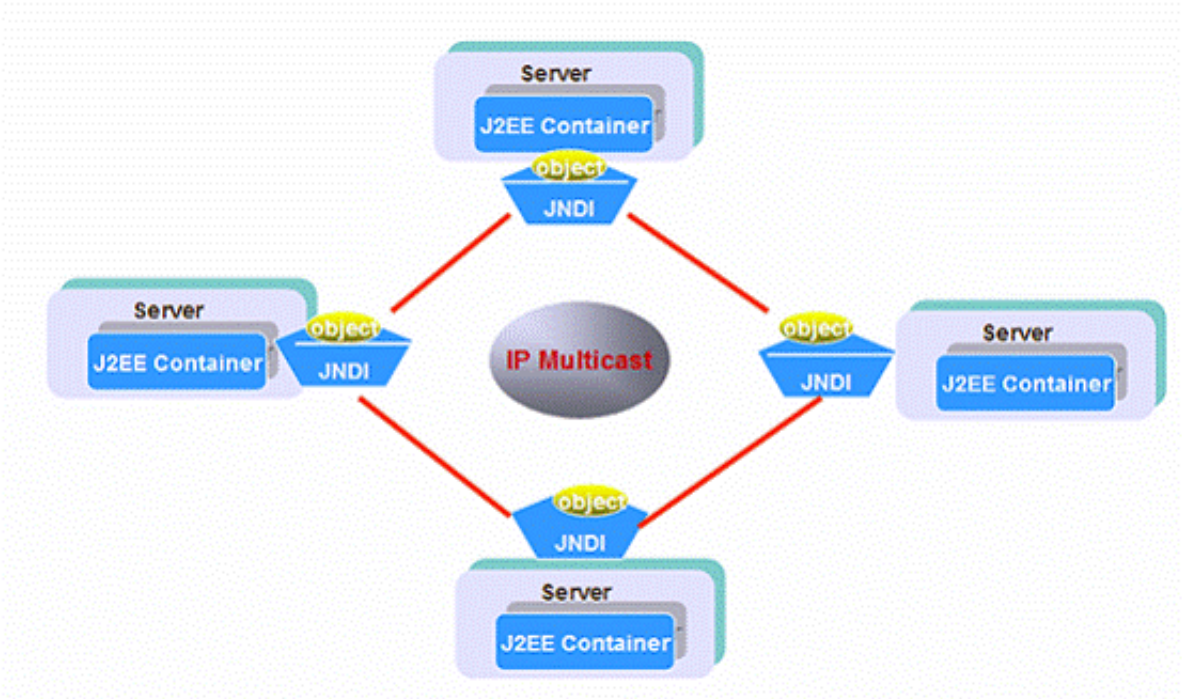


Figure 14:Shared global JNDI

Shown as figure 14, the shared global JNDI tree actually consists of all the local JNDI deposits in every node. Each node in a cluster hosts its own JNDI naming server, which replicate everything to the other naming servers in the cluster. Thus, every naming server has a copy of every other naming server's objects in the tree. This redundant structure makes the Global JNDI Tree highly available.

In practice, this clustered JNDI tree can be used for two purposes. You can use it for deployment which is the task of administrator. After deploying EJB modules or setting JDBC&JMS services in one server instance, all the objects in the JNDI tree will be replicated to other instances. During runtime of applications, your programs can access JNDI tree to store and retrieve objects by using JNDI API, and your custom objects are also be replicated globally.

Independent JNDI

While JBoss and Weblogic all adopt global shared JNDI, Sun JES, IBM Websphere and others utilize an independent JNDI tree for each application server. Member servers in an independent JNDI tree cluster do not know or care about the existence of other servers in the cluster. Does this mean they don't want clustered JNDI? As almost any EJB access starts with a lookup of its home interface in a JNDI tree, the clustering features would be almost useless without a clustered JNDI tree.

Actually, the independent JDNI tree can still have highly available features only if their J2EE application is homogeneous. We call it a homogeneous cluster when all the instances in the cluster have the same settings and have deployed the same set of applications. Under such condition, special admin tools called agent can help achieve the high availability, shown as figure 15.

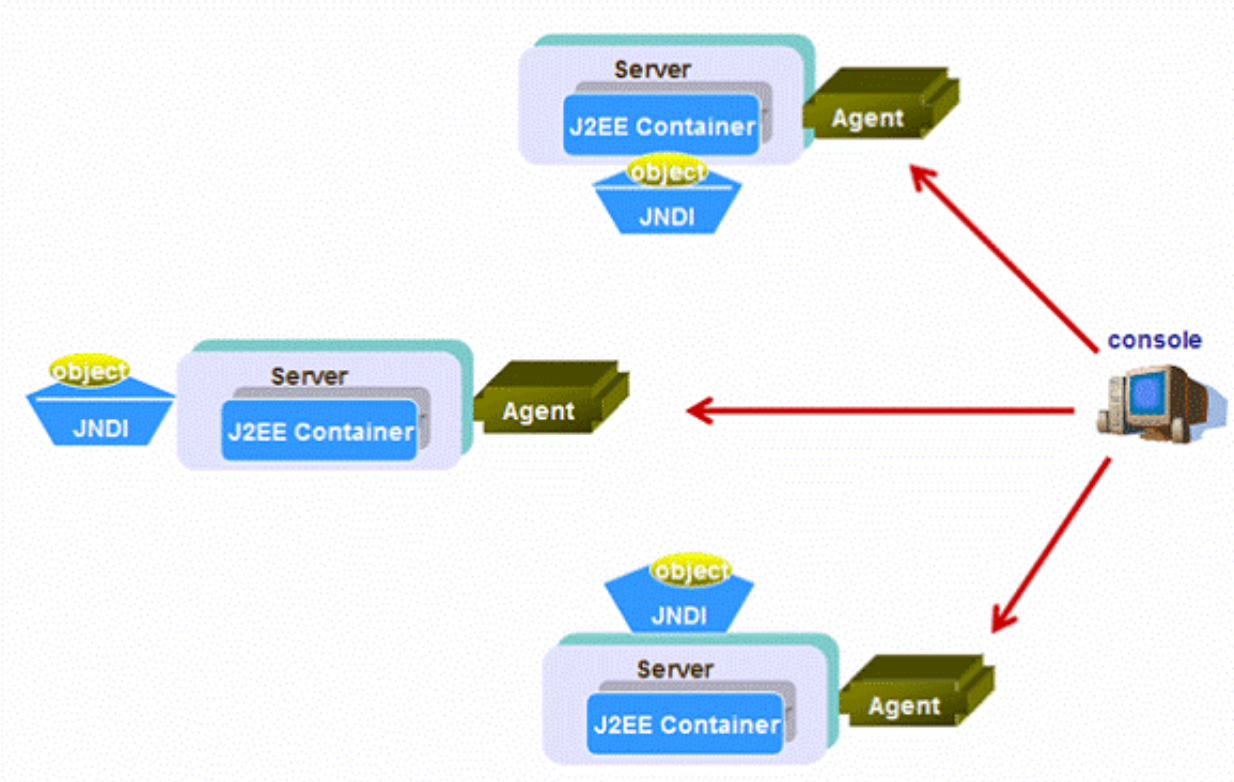


Figure 15: Independent JNDI

Both Sun JES and IBM Websphere have node agent installed on each instance in the cluster. When

deploy EJB modules and binding other JNDI services, the admin console can send commands to all agents to achieve the same effect of the global shared JNDI tree.

But the independent JNDI solution will not support replication for arbitrary objects which are bound and retrieved by running applications. They have reasons for this: the primary role of JNDI in a J2EE application is to provide the indirection layer for administrating external resources, not for runtime data deposits. If such requirements happen, an individual LDAP server or database with HA features can help. Both Sun and IBM have their own individual LDAP server products which are already shipped with clustering features.

Centralized JNDI

A few of J2EE products use centralized JNDI tree solution in which the naming server is hosted on a single server and all servers instances register their same EJB components and other admin objects on the single naming server.

The naming server itself implements highly available features which is transparent to client. All clients look up EJB components in this single naming server. But this structure always implies complex installation and administration and has thus been abandoned by most vendors.

Initial access to JNDI server

When clients to access the JNDI server, they need to know the hostname/IP address and port number of the remote JNDI server. In global and independent JNDI tree solutions, there are more than one JNDI servers. Which one should clients connect to for the first access to JNDI servers? How to achieve load balancing and failover?

Technically, a software or hardware load balancer can sit between the remote clients and all JNDI servers to perform load balancing and failover. But few vendors implement this way, there are many simple solutions.

- Both Sun JES and JBoss implement JNDI clustering by making the “java.naming.provider.url” JNDI setting accept a list of URLs separated by a comma. For example,
  java.naming.provider.url=server1:1100,server2:1100,server3:1100,server4:1100
  The client will try to get in touch with each server from the list, one after the other, stopping as soon as one server has been reached.
- JBoss also has implemented auto-discovery features. When property string “java.naming.provider.url” is empty, the client will try to discover a bootstrap JNDI server through a multicast call on the network.

EJB clustering implementation

EJB is an important part of J2EE technology and EJB clustering is the biggest challenge when implement J2EE clustering.

EJB technology is born for distributed computing. They can be running in independent servers. Web server components or rich clients can access the EJBs from other machines through standard protocol (RMI/IIOP). You can invoke methods on remote EJB just as you would invoke a method on any local

Java object. In fact, RMI-IIOP completely masks whether the object you're invoking on is local or remote. This is called *local/remote transparency*.

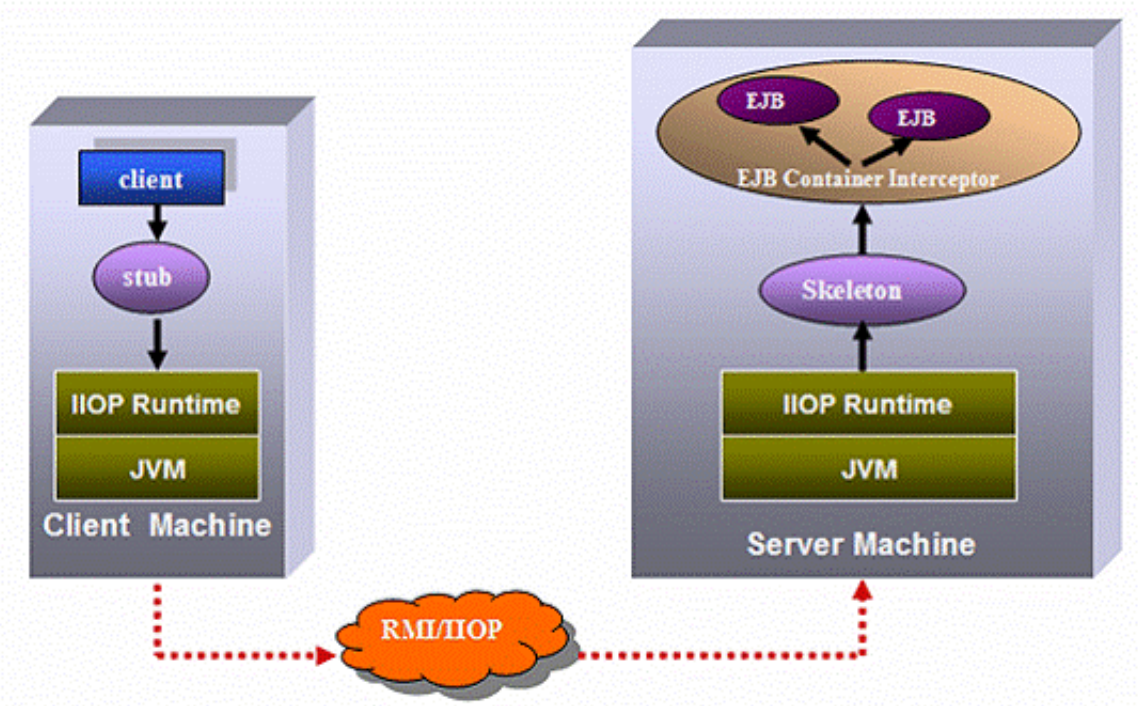


Figure 16: EJB Invoking Mechanism

The above figure shows the mechanism of invoking remote EJB. When a client wants to use an EJB, it cannot invoke the EJB directly. Instead, the client can only invoke a local object called *stub,* which acts as a proxy to the remote object and has the same interface as the remote one. The stub is responsible for accepting method calls locally and *delegating* those method calls to the remote EJBs across the network. Stubs are running within the client JVM, and know how to look over the network for the real object through RMI/IIOP. For detail information about EJB, please refer to "http://java.sun.com/products/ejb/".

Let's look at how we use EJB in our J2EE code to explain the implementation of EJB Clustering. To make a call to an EJB, you should

- Look up the EJBHome stub from a JNDI server.
- looks up or create an EJB object using the EJBHome stub; an EJBObject stub returns.
- makes method calls against the EJB using the EJBObject stub

Load balancing and failover can happen during JNDI lookup (the 1 st step), which I have already mentioned it in last section. During methods call to EJB stubs (include EJBHome and EJBObject), vendors implement EJB load balancing and failover in following three different ways.

Smart stub

As you know, client can access the remote EJB through a stub object, this object can be retrieved from a JNDI tree, and it is even possible that clients download the classfile of the stub from any web server transparently. So the stub has the following features:

It can be generated dynamically and programmatically at runtime and the definition of the stub (the classfile) does not necessary needs to be in the classpath of client environment or part of the client

libraries (JAR) at runtime (as it can be downloaded).

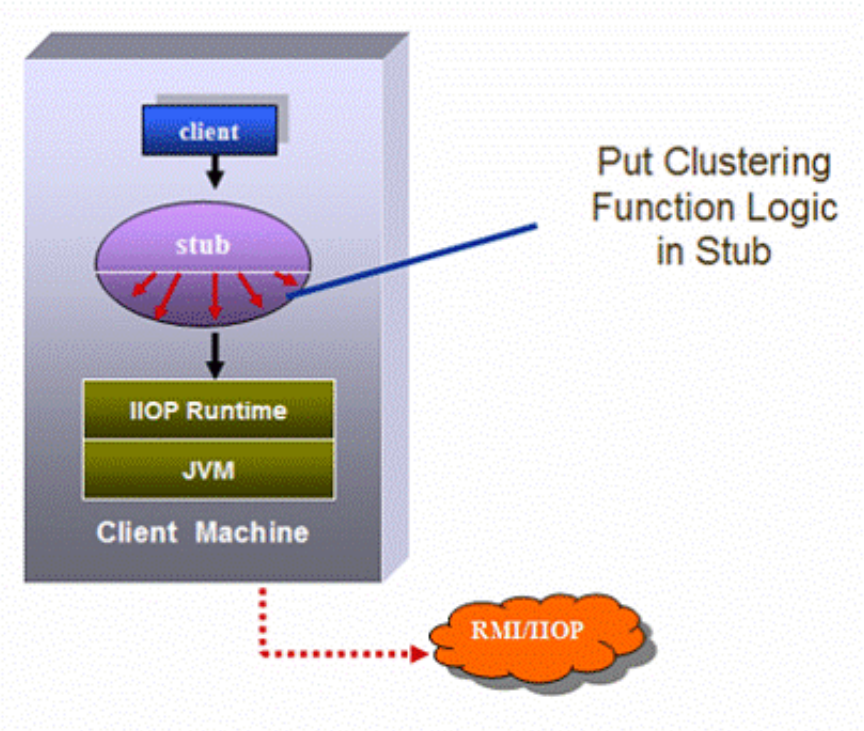


Figure 17: Smart Stub

Shown as figure 17, BEA Weblogic and JBoss implement EJB clustering by incorporating some specific behavior, in the stub code, that will transparently run on the client side (the client code doesn't even know about this code). This technique is called "smart stub".

The smart stub is really smart that it can contain the list of target instances it can access, it can detect any failure about the target instances, and it also contains complex load-balancing and fail-over logic to dispatch requests to the targets. Furthermore, if the cluster topology changes (for example: new instances added in or removed off), the stub can update itself of the target list to reflect the new topology without manually reconfiguration.

Put the clustering implementation in the stub has following advantages:

- Since EJB stub is running inside the client environment, it will save a lot of resources in the server side.
- The load balancer is incorporated in the client code and is highly related with client life cycles. This will eliminate single point of failure of load balancer. If the load balancer dies, it most probably means that the client code is also dead, which is acceptable.
- The stub can be downloaded dynamically and update itself automatically. That means zero maintenance.

IIOP Runtime Library

Sun JES Application Server implements EJB clustering in another way. The load balancing and failover logic are implemented in the IIOP runtime library. For example, JES has modified the "ORBSocketFactory" implementation to let it be cluster-aware, shown as figure 18.

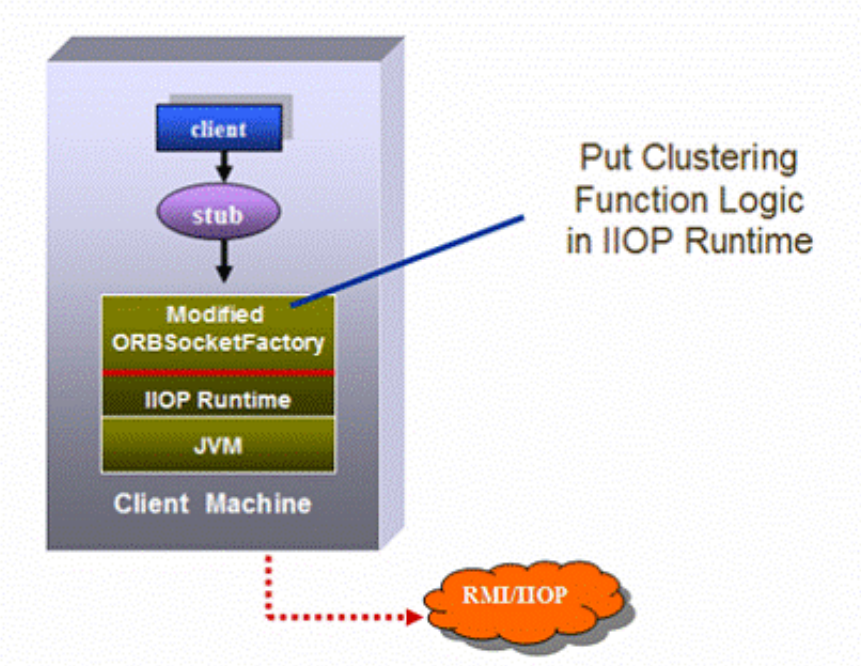


Figure 18: IIOP Runtime

The modified version of "ORBSocketFactory" has all the logics and algorithms to perform load balancing and failover, which will keep the stub small and clean. Since the implementation is in the runtime library, it can get system resources more easily than stub does. But this approach always requires the specific runtime library in the client side, which will make some troubles when interoperating with other J2EE products.

Interceptor Proxy

IBM Websphere employs a Location Service Daemon (LSD), which acts as a interceptor proxy to EJB clients, shown as figure 19.

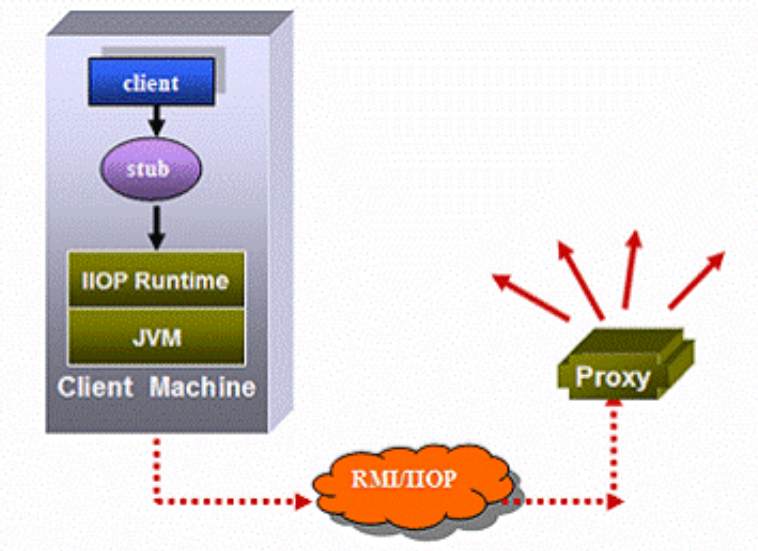


Figure 19: Interceptor Proxy

Within this approach, a client obtains a stub by looking up from JNDI. The stub contains routing

information to the LSD rather than to the application server which hosts EJBs. Then the LSD receives all coming requests and determines where to send them to different instances based on the load balancing and failover policy. This approach will add extra administration works to install and maintain the cluster.

Clustering support for EJBs

To invoke a method of an EJB, two types of stub objects are involved: one for the EJBHome interface and one for the EJBObject interface. This means that EJBs can potentially realize the load balancing and failover on two levels:

- When a client create or looks up an EJB object using the EJBHome stub
- When a client makes method calls against the EJB using the EJBObject stub

Clustering support for EJBHome Stub

The EJBHome Interface is used to create or lookup EJB instances in the EJB container and EJBHome Stub is the client agent for EJBHome Interface. EJBHome interface will not maintain any state information for the client. So, the same EJBHome interface from different EJB containers is identical for the client. When the client issues a create() or find() call, the home stub selects a server from the replica list in accordance with the load balancing and failover algorithm, and routes the call to the home interface on that server.

Clustering support for EJBObject Stub

When an EJBHome interface creates an EJB instance, it returns an EJBObject stub to the client to let the user make business methods call to the EJB. The system already has a list of all of the available servers in the cluster, to which the beans are deployed, but it cannot route the calls issued by the EJBObject stub to the EJBObject interface on arbitrary server instance, depend on the EJB type.

Stateless session bean is most probably the easiest case: as no state is involved, All EJB instances are considered identical. So the method invoking from EJBObject can be load-balanced or failed over on any participating server instances.

Stateful Session Beans are clustered a bit differently from the stateless beans. As you know, Stateful Session Beans will hold session state information for a client in successive requests. Technically, clustering of Stateful Session Beans is the same as clustering of HTTPSession. At normal time, the EJBObject stub will not load balance the requests to different server instances. Instead, it will stick to the instance where the EJBObject is created at first time; we call this instance the “primary instance”. During processing, the state information will backup from the primary instance to other servers. If the primary instance fails, other backup servers will take over.

Entity Beans are stateless essentially, although they can process Stateful requests. All the information data are backed up into database by the mechanism of Entity Bean itself. It seems that for Entity Beans, load balancing and failover can be achieved easily just like Stateless Session Bean. But actually, Entity Beans are not load balanced and fail-overed most of time. As suggested by design patterns, entity beans are always wrapped by a session bean façade. Therefore, most access to entity EJBs should occur over local interfaces by in-process session beans, rather than remote clients. This will make load balancing and failover become no sense.

Clustering support for JMS and database connection

There are other distributed objects in J2EE in addition to JSP, Servlet, JNDI and EJB. These objects may or may not be supported in a cluster implementation.

Currently, some database products, such as Oracle RAC, support clustering environment and can deploy multi replicated, synchronized database instances. However, JDBC is a highly stateful protocol and in which transactional state are tied directly to the socket between the server and the client, so it is hard to achieve clustering. If a JDBC connection dies, a ll JDBC objects associated with the dead connection will also be defunct. The re-connection action is needed in the client code. BEA Weblogic uses a JDBC multipool to eases the reconnection process.

JMS is supported in most of J2EE servers, but not fully supported. Load balancing and failover is implemented only for JMS broker, and few products have the failover functions for messages in JMS Destinations.

Myths about J2EE clustering

Failover can avoid errors completely. -- Negative

In the document of JBoss, there is a whole section to warn you “do you really need HTTP sessions replication?” Yes, sometime a high availability solution without failover is acceptable and cheap. And further more, the failover feature is not as strong as you expected.

What on earth does failover bring to you? Some of you may think that failover can avoid errors. You see, without session failover, session data is lost when a server fails and causes errors; while with session failover, sessions can be restored from the backup and requests can be processed by another server instance, the client even isn't aware of the failure. That may be true, but it’s conditional!

Remind that when I defined “failover”, I defined a condition for when the failover will happen: “between the method calls”. It means if you have two successive methods to a remote object, the failover will only happen after the first method call is finished successfully and before the second method request is sent out.

So, what will happen if the remote server fails when the methods are in the middle of processing in the server? The answer is: the process will stop and the client will see error messages in most cases, unless the methods are *idempotent (defined in the “basic terminology” section)*.Only if the methods are *idempotent,* some load balancers are smart enough to retry these methods and failover them to other instances.

Why is “*idempotency” important*? Because the client never knows where the execution the request was in when the failure occurred. Has the method just been initiated or it is almost finished? A client can never determine it! If the method is not idempotent, two invokings of the same method will alter the system state twice and the system will be in an inconsistent situation.

You might think that all methods that are placed in a transaction are idempotent. After all, if failure

happens, the transaction will roll back, and all transactional state changes will be undone. But the truth is that the transaction boundary may not include all the edges of remote methods invoking. What if the transaction commits on the server and the network crashes on the return trip to the client. The client would not know whether the server's transaction succeeded or not.

In serious applications, to make all the methods *idempotent is* impossible. So, you can only reduce errors by failover, but not avoid them! Take an online store website for example, suppose every server instance will handle 100 online users' request at any time. When one server fails, the solution without session failover will lose all the users' session data and anger all the 100 users; while in the solution with session failover, only 20 users' requests are in processing when the server fails and only these users are angry about the failure. All the other 80 users are just in the thinking time or between the methods. These users get their session failed over transparently. So, you should trade off from following considerations:

- The different impact between anger 20 users and 100 users.
- The different cost between products with failover and without failover.

Stand-alone applications can be transmit transparently to a cluster structure. -- Negative!

Although some vendors announce such flexibility for their J2EE products, don't trust them! Actually, you should prepare for the clustering at the beginning of system design and impact all the phases including development and testing.

Http Session
In a cluster environment, there are many restrictions to HTTPSession usage as I mentioned before, depending on different mechanism your application server uses for session failover. The first important restriction is that all objects stored in the HTTPSession must be serializable which will limit the design and structure of the application. Some design patterns and MVC framework will use HTTPSession to store some non-serializable objects (such as Servlet context, Local EJB Interface and web services reference), such designs cannot work in a cluster. Secondly, object serialization and de-serialization are very costly in performance especially in the database persistent approach. In such environment, storing large or numerous objects in the session should be avoided. If you have chosen a memory replication approach, be careful about the limitation on cross-referenced attributes in HTTPSession as I mentioned before. Another major difference in cluster environment is you are required to call "setAttribute ()" method whenever any attribute under HTTPSession is modified. This method is optional in stand alone system. The purpose of this method is to separate modified attributes from those untouched, so that the system can backup only necessary data for session failover to improve performance.

Cache
Almost every J2EE project I experienced used object caching to improve performance, and all popular application servers provide extra degrees of caching to enable faster applications. But these caches are typically designed for a standalone environment, and can only work within one JVM instance. We need cache because some objects are so heavy that creating a new one will cost much. So we maintain an object pool to reuse the object instances without further creation. We gain performance only if the maintenance of the cache is cheaper than objects creation. In a clustered environment, each JVM instance will maintain its own copy of the cache, which should be synchronized with others to provide inconsistent state in all server instances. Sometimes this kind of sync will bring worse performance than no caching at all.

Static variables

When design J2EE applications, design patterns are popular among architects. Some design pattern such as “Singleton” will use a static variable to share a state among many objects. This approach works well on a single server, but fails in a cluster. Each instance in the cluster would maintain its own copy of the static variable in its own JVM instance, thereby breaking the mechanism of the pattern. One example for the usage of static variable is to keep statistics about total number of online users. One easy way is to store the number in a static variable, increasing and decreasing it when users are in and out. This application works absolutely fine on a single server, but fails on a cluster. A preferable way workable with cluster is to store all state data to a database.

External resource
Although not recommended by the J2EE specification, the external I/O operations are used for various purposes. For example, some applications use file systems to save uploading files by users, or create dynamic configuration XML files. In a cluster the application server has no way of replicating these files across to other instances. To work in a cluster, the solution is to use the database in place of external files, if possible. One could also choose SAN as central deposits for files.

Special Services
There are some special services which only make sense in the stand-alone mode. Timer services are good examples of such services, which will happen regularly and based on constant interval. Timer services are often used to perform administrative tasks automatically, such as logging file rotation, system data backup, database consistence checking and redundant data cleaning up. Some event based services are also hard to migrate to a cluster environment. The initial services are good examples which will happen only at the beginning of whole system. Email notification services are also such examples which are trigged by some warning conditions.

These services are trigged by events instead of requests, and should only be executed only once. Such services will make the load balancing and failover make little sense in a cluster.

Some products have prepared for such services. For example, JBoss uses “clustered singleton facility” to coordinate all the instances to guarantee to execute these services once and only once. Based on your product platform you choose, those special services may be an obstacle to migrate your applications to a cluster structure.

Distributed structure is more flexible than collocated one? -- Maybe Not!

J2EE technology, especially EJB, is born for distributed computing. Decoupled business functionality, reused remote components make multi-tier applications popular. But we won’t make everything distributed. Some J2EE architects think it better that the Web tier and EJB tier should be collocated closely. These kinds of discussion are still going on. Let me explain more.

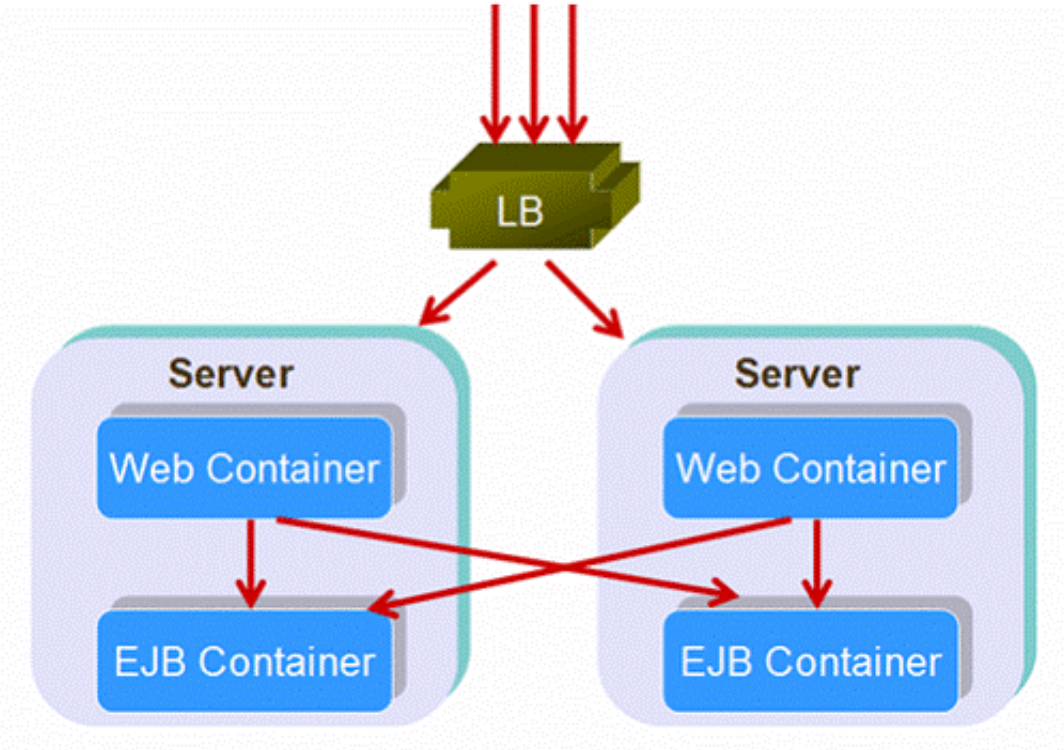


Figure 20: Distributed Structure

Shown as figure 18, it is a distributed structure. When requests come, load balancer will dispatch them to different web containers in different servers. If the requests include EJB invokes, the web container will re-dispatch the EJB invokes to different EJB containers. Such, requests are load balanced and failed over twice.

Some people look down on the distributed structure. They have pointed out:

- The second load balancing is not necessary, because it cannot assign tasks more evenly. Every server instance will has its own web container and EJB container. To make the EJB container to process requests from other instance's web container shows no advantages compared to inner invoking happened inside server instances.
- The second failover is not necessary, because it cannot improve availability. Most vendors implement their J2EE servers in such a way that web container and EJB container within the same server share the same JVM instance. If the EJB container fails, under most circumstances, the Web container in the same JVM instance will also fail at the same time.
- Performance will degrade. Imagine now in one method of your application you may be invoking a couple of EJBs, if you load balance on every one of those EJBs, you're going to end up with instances of your application spread out across the multi server instances; you're going to have a lot of server-to-server cross talk that's unnecessary. And more, if your method is under a transaction, your transaction boundary will include many server instances which will impact performance heavily.

At the runtime actually, most vendors (include Sun JES, Weblogic and JBoss) will optimize the EJB load balancing mechanism to let requests first choose the EJB container in the same server. In this way, shown as figure 19, we load balance only at the first level of requests (web container), and then have subsequent services end up on that same server. This structure is called collocated structure. Technically, collocated structure is the special case of distributed one.

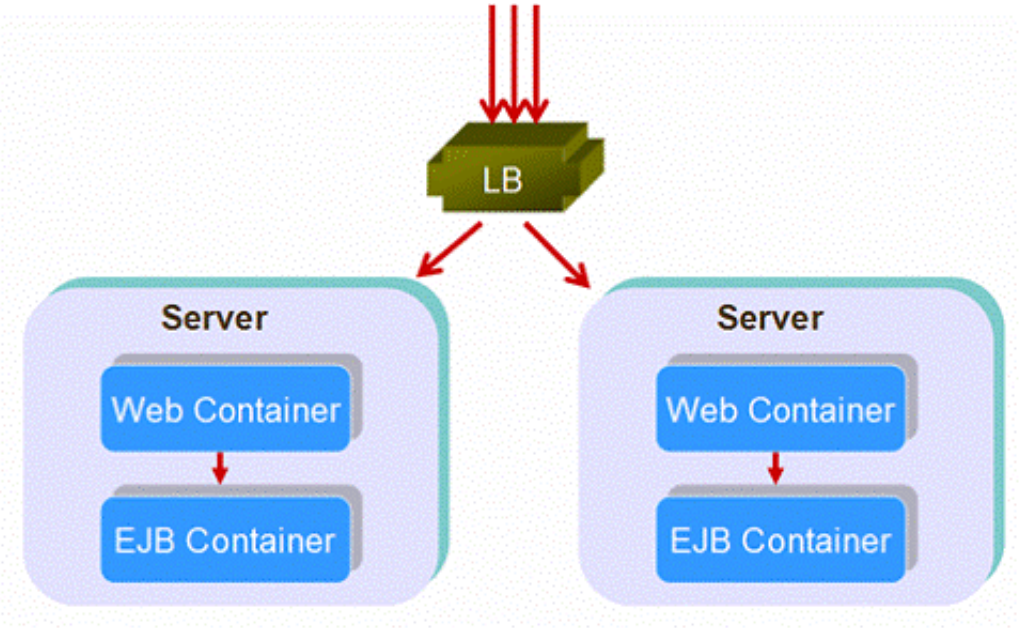


Figure 21: Collocated Structure

One interesting question is that, since most deployment are evolved as collocated structure at the runtime, why not use local interface instead of remote interface, and this will improve performance quite a bit. Of course you can. But remember, when using local interface, Web components and EJB are coupled tightly, and make method invoking directly instead of IIOP/RMI. The load balancer and failover dispatcher have no chance to intervene with local interface call, the "Web+EJB" process is load balanced and failover as a whole.

But unfortunately, using local interface in a cluster has some limitations on most J2EE servers. EJBs are local objects with local interfaces, but they are not Serializable. So the limit is that the local references are not allowed to be stored in HTTPSession. Some products, such as Sun JES, treat local EJBs differently and make them Serializable and can be used in HTTPSession as you will.

Another interesting question is: Since collocated structure is popular and has good performance, why we need distributed structure? Like in most cases, things happen for a reason. Sometime, the distributed structure is not replaceable:

- EJB is not only for web container, rich clients are also the consumers.
- EJB components and Web components may be in different security levels, and need to be separated physically. So, firewall can be setup to protect the most important machines on which EJB components are running.
- Extreme unsymmetricalness between Web and EJB tier will make the distributed structure a better choice. For example, some EJB components are so complex and resource consuming, that they can only run in some expensive big servers; on the other hand, the Web components (html, JSP and Servlet) are simple enough that cheap PC servers will be satisfied. Under such condition, dedicated Web servers will be used to accept client connection requests, and serve static data (HTML and images) and simple Web components (JSP and Servlet) very quickly. The big servers are only used for complex computing and make the best use of the investment.

Conclusion

Clustering is different from the stand-alone environment. J2EE vendors implement clustering differently. You should prepare for J2EE clustering at the beginning of your projects in order to build a large scale system. Choose proper J2EE product which is well suitable to your requirements. Choose proper third-party software and frameworks to make sure they are cluster-aware too. Then, design proper architectures which will really benefit from clustering instead of suffering.

About the author

Wang Yu presently works for GPE group of Sun Microsystems as a Java technology engineer and technology architecture consultant. His duties include supporting local ISVs, evangelizing and consulting on important Java technologies such as J2EE, EJB, JSP/Servlet, JMS, Web services technologies. He can be reached at yu.wang@sun.com.