

Technical Article

Session Management for Clustered Applications

by Jon Purdy

Originally published on BEA Dev2Dev February 2005

Abstract

Clustered applications require reliable and performant HTTP session management. Unfortunately, moving to a clustered environment introduces several challenges for session management. This article discusses those challenges and proposes solutions and recommended practices. The included session management features of BEA WebLogic Server and of Tangosol Coherence*Web are examined here.

Basic Terminology

An HTTP session ("session") spans a sequence of user interactions within a web application. "Session state" is a collection of user-specific information. This session state is maintained for a period of time, typically beginning with the user's first interaction and ending a short while after the user's last interaction, perhaps thirty minutes later. Session state consists of an

arbitrary collection of "session attributes," each of which is a Java object and is identified by name. "Sticky load alancing" describes the act of distributing user requests across a set of servers in such a way that requests from a given user are consistently sent to the same server.

Coherence is a data management product from Tangosol that provides real-time, fully coherent data sharing for clustered applications. Coherence*Web is a session management module that is included as part of Coherence. An HTTP session model ("session model") describes how Coherence*Web physically represents session state. Coherence*Web includes three session models. The Monolithic model stores all session state as a single entity, serializing and deserializing all attributes as a single operation. The Traditional model stores all session state as a single entity but serializes and deserializes attributes individually. The Split model extends the Traditional model but separates the larger session attributes into independent physical entities. The applications of these models are described in later sections of this article.

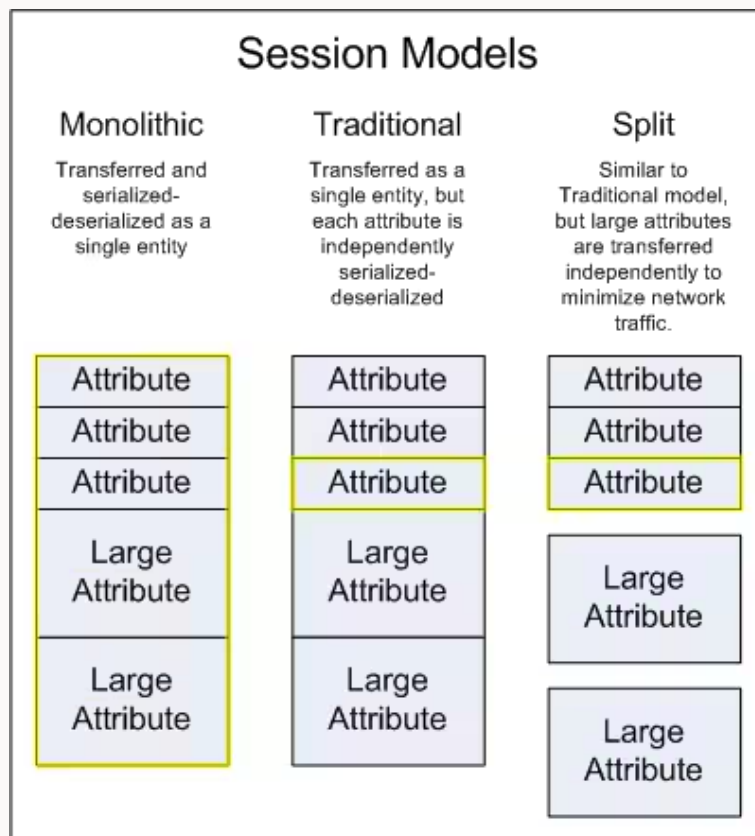


Figure 1. Session models

Sharing Data in a Clustered Environment

Session attributes must be serializable if they are to be processed across multiple JVMs, which is a requirement for clustering. It is possible to make some fields of a session attribute non-

clustered by declaring those fields as `transient`. While this eliminates the requirement for all fields of the session attributes to be serializable, it also means that these attributes will not be fully replicated to the backup server(s). Developers who follow this approach should be very careful to ensure that their applications are capable of operating in a consistent manner even if these attribute fields are lost. In most cases, this approach ends up being more difficult than simply converting all session attributes to serializable objects. However, it can be a useful pattern when very large amounts of user-specific data are cached in a session.

The J2EE Servlet specification (versions 2.2, 2.3, and 2.4) states that the servlet context should not be shared across the cluster. WebLogic Server implements this specification as stated. Non-clustered applications that rely on the servlet context as a singleton data structure will have porting issues when moving to a clustered environment. Coherence*Web does support the option of a clustered context, though generally it should be the goal of all development teams to ensure that their applications follow the J2EE specifications.

A more subtle issue that arises in clustered environments is the issue of object sharing. In a non-clustered application, if two session attributes reference a common object, changes to the shared object will be visible as part of both session attributes. However, this is not the case in most clustered applications. To avoid unnecessary use of compute resources, most session management implementations serialize and deserialize session attributes individually on demand. Both WebLogic Server and Coherence*Web (Traditional and Split session models) normally operate in this manner. If two session attributes that reference a common object are separately deserialized, the shared common object will be instantiated twice. For applications that depend on shared object behavior and cannot be readily corrected, Coherence*Web provides the option of a Monolithic session model, which serializes and deserializes the entire session object as a single operation. This provides compatibility for applications that were not originally designed with clustering in mind.

Many projects require sharing session data between different web applications (and with BEA WebLogic Portal, possibly between different portlets). The challenge that arises is that each web application typically has its own class loader. As a result, objects cannot readily be shared between separate web applications. There are two general methods for working around this, each with its own set of trade-offs. Developers can place common classes in the Java CLASSPATH, allowing multiple applications to share instances of those classes at the expense of a slightly more complicated configuration. An alternative is to use Coherence*Web to share session data across class loader boundaries. Each web application is treated as a separate cluster member, even if they are running within the same JVM. This approach provides looser coupling between web applications (assuming serialized classes share a common `serialVersionUID`), but suffers from a performance impact because objects must be serialized-deserialized for transfer between cluster members.

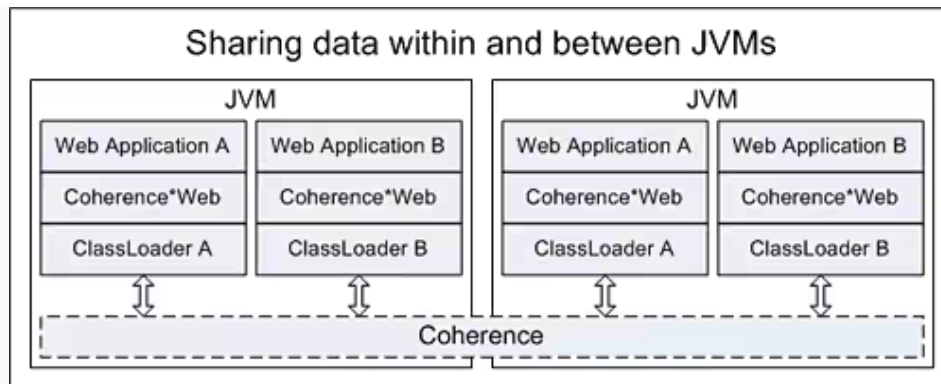


Figure 2. Sharing data between web applications or portlets by clustering (serializing-deserializing session state)

Reliability and Availability

An application must guarantee that a user's session state is properly maintained in order to exhibit correct behavior for that user. Some availability considerations occur at the application design level and apply to both clustered and non-clustered applications. For example, the application should ensure that user actions are idempotent: The application should be capable of handling a user who accidentally submits an HTML form twice. (Additionally, WebLogic Server can automatically fail over idempotent HTTP requests if a server fails in the middle of request processing.)

With sticky load balancing, issues related to concurrent session updates are normally avoided, as all updates to session state are made from a single server (which dramatically simplifies concurrency management). This has the benefit of ensuring no overlap of user requests occurs even in cases where a user submits a new request before the previous request has been fully processed. Use of HTML frames complicates this, but the same general pattern applies: Simply ensure that only one display element is modifying session state.

In cases where there may be concurrent requests, Coherence*Web manages concurrent changes to session state (even across multiple servers) by locking sessions for exclusive access by a single server. With Coherence*Web, developers can specify whether session access is restricted to one server at a time (the default), or even one thread at a time.

As a general rule, all session attributes should be treated as immutable objects if possible. This ensures that developers are consciously aware when they change attributes. With mutable objects, modifying attributes often requires two steps: modifying the state of the attribute object, and then manually updating the session with the modified attribute object by calling `setAttribute()`. This means that your application should always call `setAttribute()` if the attribute value has been changed, otherwise, WebLogic Server will not replicate the

modified attribute value to the backup server. Coherence*Web tracks all mutable attributes retrieved from the session, and so will automatically update these attributes, even if `setAttribute()` has not been called. This can help applications that were not designed for clustering to work in a clustered environment.

With WebLogic Server, session state is normally maintained on two servers, one primary and one backup. A sticky load balancer will send each user request to the specified primary server, and any local changes to session state will be copied to the backup server. If the primary server fails, the next request will be rerouted to the backup server, and the user's session state will be unaffected. While this is a very efficient approach (among other things, it ensures that the cluster is not overwhelmed with replication activity after a server failure), there are a few drawbacks. Because session state is copied when the session is updated, failure (or cycling) of both the primary and backup servers between session updates will result in a loss of session state. To avoid this problem, wait thirty minutes between each server restart when cycling a cluster of WebLogic Server instances. The thirty-minute interval increases the odds of a return visit from a user, which can trigger session replication. Additionally, if the interval is at least as long as the session timeout, the session state will be discarded anyway if the user has not returned.

This cycling interval is not required with Coherence*Web, which will automatically redistribute session data when a server fails or is cycled. Coherence's "location transparency" ensures that node failure does not affect data visibility. However, node failure does impact redundancy, and therefore fresh backup copies must be created. With most Coherence*Web configurations, two machines (primary and backup) are responsible for managing each piece of session data, regardless of cluster size. With this configuration, Coherence can handle one failover transition at any given point in time. When a server fails, no data will be lost as long as the next server failure occurs after the completion of the current failover process. The worst-case scenario is a small cluster with large amounts of session data on each server, which may require a minute or two to rebalance. Increasing the cluster size, or reducing the amount of data storage per server, will improve failover performance. In a large cluster of commodity servers, the failover process may require less than a second to complete. For particularly critical applications, increasing the number of backup machines will increase the number of simultaneous failures that Coherence can manage.

The need for serialization in clustered applications introduces a new opportunity for failure.

ORACLE

Products Industries Resources Customers Partners Developers Company



View Accounts



Contact Sales

overflow.

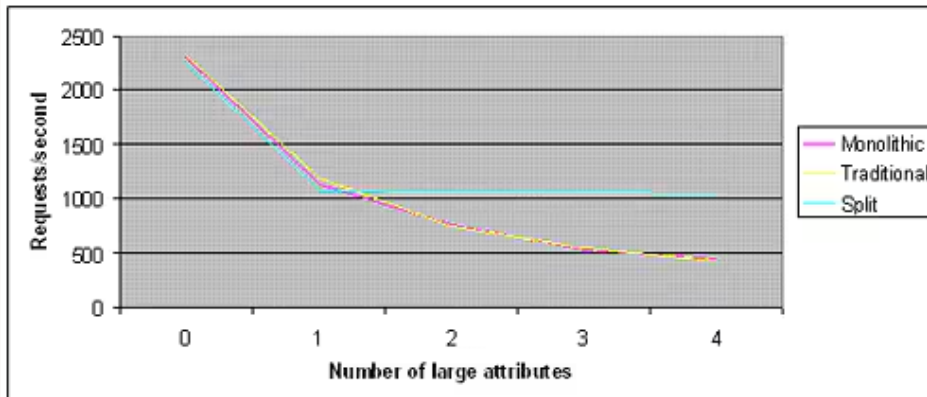


Figure 3. Performance as a function of session size. Each session consists of ten 10-character Strings and from zero to four 10,000-character Strings. Each HTTP request reads a single small attribute and a single large attribute (for cases where there are any in the session), and 50 percent of requests update those attributes. Tests were performed on a two-server cluster. Note the similar performance between the Traditional and Monolithic models; serializing-deserializing Strings consumes minimal CPU resources, so there is little performance gain from deserializing only the attributes that are actually used. The performance gain of the Split model increases to over 37:1 by the time session size reaches one megabyte (100 large Strings). In a clustered environment, it is particularly true that application requests that access only essential data have the opportunity to scale and perform better; this is part of the reason that BEA suggests keeping sessions to a reasonable size.

Another optimization is the use of transient data members in session attribute classes. Because Java serialization routines ignore transient fields, they provide a very convenient means of controlling whether session attributes are clustered or isolated to a single cluster member. These are useful in situations where data can be "lazy loaded" from other data sources (and therefore recalculated in the event of a server failover process), and also in scenarios where absolute reliability is not critical. If an application can withstand the loss of a portion of its session state with zero (or acceptably minimal) impact on the user, then the performance benefit may be worth considering. In a similar vein, it is not uncommon for high-scale applications to treat session loss as a session timeout, requiring the user to log back in to the application (which has the implicit benefit of properly setting user expectations as to the state of their application session).

Sticky load balancing plays a critical role with WebLogic Server because session state is not globally visible across the cluster. For high-scale clusters, user requests normally enter the application tier through a set of stateless load balancers, which redistribute (more or less randomly) these requests across a set of sticky load balancers, such as Microsoft IIS or Apache HTTP Server with the WebLogic Server plugin. These sticky load balancers are responsible for the more computationally intense act of parsing the HTTP headers to determine which WebLogic Server instance will be processing the request (based on the server ID specified by the session cookie). If requests are misrouted for any reason, session integrity will be lost. For example, some load balancers may not parse HTTP headers for requests with large amounts of POST data (for example, more than 64KB), so these requests will not be routed to the appropriate WebLogic Server instance. Other causes of routing failure include corrupted or malformed server IDs in the session cookie. Most of these issues can be handled with proper selection of a load balancer as well as designing tolerance into the application whenever possible (for example, ensuring that all large POST requests avoid accessing or modifying session state).

Sticky load balancing aids the performance of Coherence*Web but is not required. Because Coherence*Web is built on the Coherence data management platform, all session data is

globally visible across the cluster. A typical Coherence*Web deployment places session data in a near cache topology, which uses a partitioned cache to manage huge amounts of data in a scalable and fault-tolerant manner, combined with local caches in each application server JVM to provide instant access to commonly used session state. While a sticky load balancer is not required when Coherence*Web is used, there are two key benefits to using one. Due to the use of near cache technology, read access to session attributes will be instant if user requests are consistently routed to the same server, as using the local cache avoids the cost of deserialization and network transfer of session attributes. Additionally, sticky load balancing allows Coherence to manage concurrency locally, transferring session locks only when a user request is rebalanced to another server.

Conclusion

Clustering can boost scalability and availability for applications. Clustering solutions such as WebLogic Server and Coherence*Web solve many problems for developers, but successful developers must be aware of the limitations of the underlying technology, and how to manage those limitations. Understanding what the platform provides, and what users require, gives developers the ability to eliminate the gap between the two.

Additional Reading

- Using WebLogic Server Clusters: Replication and Failover for Servlets and JSPs
- Using WebLogic Server Clusters: Load Balancing for Servlets and JSPs

Jon Purdy is co-founder of Tangosol, a company which provides in-memory caching and data management software solutions that deliver extreme scalable performance and reliability to clustered J2EE applications.

Resources for	Why Oracle	Learn	News and Events	Contact Us
Careers	Analyst Reports	What is cloud computing?	News	US Sales: +1.800.633.0738
Developers	Best cloud-based ERP	What is CRM?	Oracle CloudWorld	How can we help?
Investors	Cloud Economics	What is Docker?	Oracle CloudWorld	Subscribe to emails
Partners	Social Impact	What is Kubernetes?	Tour	Integrity Helpline

Students and Educators

Culture and Inclusion

What is Python?

Oracle Health Summit

Security Practices

What is SaaS?

Oracle DevLive

Search all events

© 2024 Oracle

[Privacy / Do Not Sell My Info](#)

[Cookie Preferences](#)

[Ad Choices](#)

[Careers](#)

