공개 S/W 기술지원
(주)구름인터랙티브

한국소프트웨어진흥원
공개SW기술지원센터

<Revision 정보>

일자	VERSION	변경내역	작성자
2007.02.22	0.1	초기 작성	한상문

	공개SW 기술지원	
	구분 : 기술지원	단계:
	작성자: 한상문	작성일: 2007.02.26
	검토자:	검토일:
	승인자:	승인일:

1. 대상기업/기관 정보

구분	항목	내용	비고
기업/기관 정보	지역		
	기업/기관 명칭	(주)구름인터랙티브	
	부서	시스템운영팀	
	직책	대리	
	담당자 이름	백승일	
	전화번호 / 팩스번호	011-9143-8240	
	E-Mail		

2. 대상기업/기관 지원사항

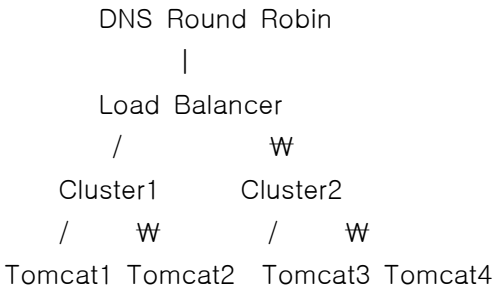
구분	항목	내용	비고
기업/기관 지원사항	접수내용	웹서버로 사용중인 여러 대의 톰캣 클러스터링에 대한 세션 리플리케이션 구성	
	지원내역	1. 톰캣 세션 클러스터링에 대한 참조 자료 송부 - http://www.onjava.com/pub/a/onjava/2004/04/14/clustering.html : 첨부파일(tomcatclustering.zip) 참조 - 톰캣 사이트(tomcat.apache.org)의 문서 송부 Clustering/Session Replication HOW-TO To run session replication in your Tomcat 5 container, the following steps should be completed: All your session attributes must implement java.io.Serializable Uncomment the Cluster element in server.xml Uncomment the Valve(ReplicationValve) element in server.xml If your Tomcat instances are running on the same machine, make sure the tcpListenPort attribute is unique for each instance. Make sure your web.xml has the <distributable/> element Load balancing can be achieved through many techniques, as seen in the Load Balancing chapter.	

Note: Remember that your session state is tracked by a cookie, so your URL must look the same from the out side otherwise, a new session will be created.

Note: Clustering support currently requires the JDK version 1.4 or later.

Overview
To enable session replication in Tomcat, three different paths can be followed to achieve the exact same thing:

Using session persistence, and saving the session to a shared file system (PersistenceManager)
Using session persistence, and saving the session to a shared database (JDBCManager)
Using in-memory-replication, using the SimpleTcpCluster that ships with Tomcat 5 (server/lib/catalina-cluster.jar)
In this release of session replication, Tomcat performs an all-to-all replication of session state. This is an algorithm that is only efficient when the clusters are small. For large clusters, the next release will support a primary-secondary session replication where the session will only be stored at one or maybe two backup servers. In order to keep the network traffic down in an all-to-all environment, you can split your cluster into smaller groups. This can be easily achieved by using different multicast addresses for the different groups. A very simple setup would look like this:



What is important to mention here, is that session replication is only the beginning of clustering. Another popular concept used to implement clusters is farming, ie, you deploy your apps only to one

server, and the cluster will distribute the deployments across the entire cluster. This is all capabilities that can go into the next release.

In the next section will go deeper into how session replication works and how to configure it.

How it Works

To make it easy to understand how clustering works, I'm gonna take you through a series of scenarios. In the scenario I only plan to use two tomcat instances TomcatA and TomcatB. We will cover the following sequence of events:

TomcatA starts up
TomcatB starts up
TomcatA receives a request, a session S1 is created.
TomcatA crashes
TomcatB receives a request for session S1
TomcatA starts up
TomcatA receives a request, invalidate is called on the session (S1)
TomcatB receives a request, for a new session (S2)
TomcatA The session S2 expires due to inactivity.
Ok, now that we have a good sequence, I will take you through exactly what happens in the session repliction code

TomcatA starts up

Tomcat starts up using the standard start up sequence. When the Host object is created, a cluster object is associated with it. When the contexts are parsed, if the distributable element is in place in web.xml Tomcat asks the Cluster class (in this case SimpleTcpCluster) to create a manager for the replicated context. So with clustering enabled, distributable set in web.xml Tomcat will create a SimpleTcpReplicationManager for that context instead of a StandardManager. The cluster class will start up a membership service (multicast) and a replication service (tcp unicast). More on the architecture further down in this document.

TomcatB starts up

		When TomcatB starts up, it follows the same sequence as TomcatA did with one exception. The cluster is started and will establish a membership (TomcatA,TomcatB). TomcatB will now request the session state from a server that already exists in the cluster, in this case TomcatA. TomcatA responds to the request, and before TomcatB starts listening for HTTP requests, the state has been transferred from TomcatA to TomcatB. In case TomcatA doesn't respond, TomcatB will time out after 60 seconds, and issue a log entry. The session state gets transferred for each web application that has distributable in its web.xml. Note: To use session replication efficiently, all your tomcat instances should be configured the same. TomcatA receives a request, a session S1 is created. The request coming in to TomcatA is treated exactly the same way as without session replication. The action happens when the request is completed, the ReplicationValve will intercept the request before the response is returned to the user. At this point it finds that the session has been modified, and it uses TCP to replicata the session to TomcatB. Once the serialized data has been handed off to the operating systems TCP logic, the request returns to the user, back through the valve pipeline. For each request the entire session is replicated, this allows code that modifies attributes in the session without calling setAttribute or removeAttribute to be replicated. a useDirtyFlag configuration parameter can be used to optimize the number of times a session is replicated. TomcatA crashes When TomcatA crashes, TomcatB receives a notification that TomcatA has dropped out of the cluster. TomcatB removes TomcatA from its membership list, and TomcatA will no longer be notified of any changes that occurs in TomcatB. The load balancer will redirect the requests from TomcatA to TomcatB and all the sessions are current. TomcatB receives a request for session S1 Nothing exciting, TomcatB will process the request as any other request. TomcatA starts up	

		Upon start up, before TomcatA starts taking new request and making itself available to it will follow the start up sequence described above 1) 2). It will join the cluster, contact TomcatB for the current state of all the sessions. And once it receives the session state, it finishes loading and opens its HTTP/mod_jk ports. So no requests will make it to TomcatA until it has received the session state from TomcatB. TomcatA receives a request, invalidate is called on the session (S1) The invalidate is call is intercepted, and the session is queued with invalidated sessions. When the request is complete, instead of sending out the session that has changed, it sends out an "expire" message to TomcatB and TomcatB will invalidate the session as well. TomcatB receives a request, for a new session (S2) Same scenario as in step 3) TomcatA The session S2 expires due to inactivity. The invalidate is call is intercepted the same was as when a session is invalidated by the user, and the session is queued with invalidated sessions. At this point, the invalidet session will not be replicated across until another request comes through the system and checks the invalid queue. Phuuuhh! :) Membership Clustering membership is established using very simple multicast pings. Each Tomcat instance will periodically send out a multicast ping, in the ping message the instance will broad cast its IP and TCP listen port for replication. If an instance has not received such a ping within a given timeframe, the member is considered dead. Very simple, and very effective! Of course, you need to enable multicasting on your system. TCP Replication Once a multicast ping has been received, the member is added to the cluster Upon the next replication request, the sending instance will use the host and port info and establish a TCP socket. Using this socket it sends over the serialized data. The reason I choose TCP sockets is because it has built in flow control	

		and guaranteed delivery. So I know, when I send some data, it will make it there :) Distributed locking and pages using frames Tomcat does not keep session instances in sync across the cluster. The implementation of such logic would be to much overhead and cause all kinds of problems. If your client accesses the same session simultanously using multiple requests, then the last request will override the other sessions in the cluster. Cluster Architecture Component Levels: Server | Service | Engine / ₩ Cluster ReplicationValve | Manager | Session Cluster Configuration The cluster configuration is described in the sample server.xml file. What is worth to mention is that the attributes starting with mcastXXX are for the membership multicast ping, and the attributes starting with tcpXXX are for the actual TCP replication. The membership is established by all the tomcat instances are sending broadcast messages on the same multicast IP and port. The TCP listen port, is the port where the session replication is received from other members. The replication valve is used to find out when the request has been completed and initiate the replication.	

		One of the most important performance considerations is the synchronous (pooled or not pooled) versus asynchronous replication mode. In a synchronous replication mode the request doesn't return until the replicated session has been sent over the wire and reinstantiated on all the other cluster nodes. There are two settings for synchronous replication. Pooled or not pooled. Not pooled (ie replicationMode="synchronous") means that all the replication request are sent over a single socket. Using synchronous mode potentially becomes a bottleneck, You can overcome this bottleneck by setting replicationMode="pooled". What is recommended here is to increase the number of threads that handle incoming replication request. This is the tcpThreadCount property in the cluster section of server.xml. The pooled setting means that we are using multiple sockets, hence increases the performance. Asynchronous replication, should be used if you have sticky sessions until fail over, then your replicated data is not time crucial, but the request time is, at this time leave the tcpThreadCount to be number-of-nodes-1. During async replication, the request is returned before the data has been replicated. async replication yields shorter request times, and synchronous replication guarantees the session to be replicated before the request returns. The parameter "replicationMode" has three different settings: "pooled", "synchronous" and "asynchronous" 2. 톰캣 클러스터링을 위한 server.xml 편집 - 하기 설정 파일 중 ***굵은/기울인 글씨체 부분 편집***	

```
<Server port="8005" shutdown="SHUTDOWN">

<!-- Comment these entries out to disable JMX MBeans support
used for the
administration web application -->
<Listener
className="org.apache.catalina.core.AprLifecycleListener" />
<Listener
className="org.apache.catalina.mbeans.ServerLifecycleListener" />
<Listener
className="org.apache.catalina.mbeans.GlobalResourcesLifecycleL
istener" />
<Listener
className="org.apache.catalina.storeconfig.StoreConfigLifecycleLis
tener"/>

<!-- Global JNDI resources -->
<GlobalNamingResources />

<!-- Define the Tomcat Stand-Alone Service -->
<Service name="Catalina">

<!-- Define a non-SSL HTTP/1.1 Connector on port 8080 -->
<Connector port="8080" maxHttpHeaderSize="8192"
maxThreads="1024" minSpareThreads="150"
maxSpareThreads="300"
enableLookups="false" redirectPort="8443" acceptCount="100"
connectionTimeout="20000" disableUploadTimeout="true" />

<!-- Define an AJP 1.3 Connector on port 8009 -->
<Connector port="8009"
enableLookups="false" redirectPort="8443" protocol="AJP/1.3" />

<!-- Define the top level container in our container hierarchy -->
<Engine name="Catalina" defaultHost="localhost">
```

```
<!-- Define the default virtual host
Note: XML Schema validation will not work with Xerces 2.2.
-->
<Host name="localhost" appBase="webapps"
unpackWARs="true" autoDeploy="true"
xmlValidation="false" xmlNamespaceAware="false">

<Cluster
className="org.apache.catalina.cluster.tcp.SimpleTcpCluster"
managerClassName="org.apache.catalina.cluster.session.Delta
Manager"
expireSessionsOnShutdown="false"
useDirtyFlag="true"
notifyListenersOnReplication="true">

<Membership
className="org.apache.catalina.cluster.mcast.McastService"
mcastAddr="224.0.0.5"
mcastPort="46002"
mcastFrequency="500"
mcastDropTime="3000"/>

<Receiver
className="org.apache.catalina.cluster.tcp.ReplicationListener"
tcpListenAddress="192.168.1.109"
tcpListenPort="4002"
tcpSelectorTimeout="100"
tcpThreadCount="6"/>

<Sender
className="org.apache.catalina.cluster.tcp.ReplicationTransmit
ter"
replicationMode="pooled"
ackTimeout="15000"
waitForAck="true"/>
```

```
<Valve
className="org.apache.catalina.cluster.tcp.ReplicationValve"
filter=".*₩.gif;.*₩.js;.*₩.jpg;.*₩.png;.*₩.htm;.*₩.html;.*₩.c
ss;.*₩.txt;"/>

<Deployer
className="org.apache.catalina.cluster.deploy.FarmWarDeplo
yer"
tempDir="/tmp/war-temp/"
deployDir="/tmp/war-deploy/"
watchDir="/tmp/war-listen/"
watchEnabled="false"/>

<ClusterListener
className="org.apache.catalina.cluster.session.ClusterSessio
nListener"/>
</Cluster>

<Valve  className="org.apache.catalina.valves.AccessLogValve"
directory="/backup/app_log/tomcat/"
prefix="localhost_access_log."  suffix=".txt"
pattern="combined"  resolveHosts="false"/>

<Valve
className="org.apache.catalina.valves.FastCommonAccessLogValv
e"
directory="/backup/app_log/tomcat/"
prefix="localhost_access_log."  suffix=".txt"
pattern="combined"  resolveHosts="false"/>
</Host>
</Engine>
</Service>
</Server>
```