Published on **The O'Reilly Network** (http://www.oreillynet.com/)
http://www.oreillynet.com/pub/a/onjava/2002/07/17/tomcluster.html
See this if you're having trouble printing code examples

Clustering with Tomcat

by Shyam Kumar Doddavula
07/17/2002

This article describes how Web applications can benefit from clustering and presents a clustering solution that we developed for the Jakarta Tomcat Servlet Engine to provide high scalability, load-balancing, and high availability using JavaSpaces technology.

Introduction

With the increasing use of Web-based applications, scalability and availability are critical for their success. Implementing a clustering solution for the servers hosting the Web applications is a simple, cost-effective solution.

JavaSpaces technology provides a distributed shared memory model that can be used to design a clustering solution. Unlike typical distributed system solutions, which revolve around enabling remote procedure calling or exchanging messages, solutions using this technology revolve around the ability to pass objects around. These objects typically carry all of the information needed to perform a task on a remote system, including even the code as needed, using an associative, shared, distributed memory. JavaSpaces provides a simple set of write, read, and take operations, which allow processes to put an object in the space, get a copy of an object, or take out an object from space.

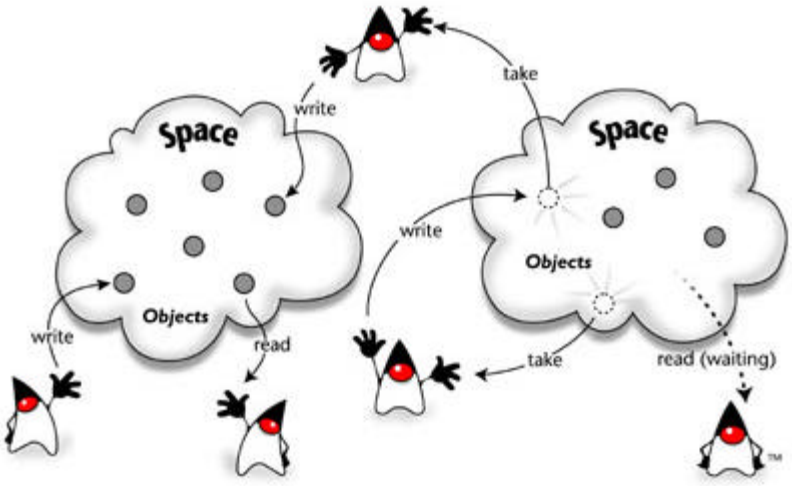

Figure 1. A distributed system using JavaSpaces (Reprinted with permission from Sun Microsystems)

JavaSpaces is a core Jini technology service. The Jini programming model, with its leasing model and dynamic service discovery, provides the infrastructure needed to create a self-configuring dynamic distributed system. See the resource section for further information on these technologies.

The JavaSpaces technology, combined with the Jini programming model, thus offers high spatial and temporal decoupling, making it possible to design a distributed system that is highly scalable, fault-tolerant, and dynamically configurable. In this article, we will describe a simple clustering solution for the Jakarta Tomcat servlet engine (Catalina 4.x), a popular open source implementation of the Servlet API, using these technologies.

Why Clustering?

Clustering solutions usually provide:

- Scalability
- High Availability
- Load Balancing

Scalability

The key question here is, if it takes time *T* to fulfill a request, how much time will it take to fulfill *N* concurrent requests? The goal is to bring that time as close to *T* as possible by increasing the computing resources as the load increases. Ideally, the solution should allow for scaling both vertically (by increasing computing resources on the server) and horizontally (increasing the number of servers) and the scaling should be linear.

High Availability

The objective here is to provide failover, so that if one server in the cluster goes down, then other servers in the cluster should be able to take over -- as transparently to the end user as possible.

In the servlet engine case, there are two levels of failover capabilities typically provided by clustering solutions:

- Request-level failover
- Session-level failover

Request-level Failover. If one of the servers in the cluster goes down, all subsequent requests should get redirected to the remaining servers in the cluster. In a typical clustering solution, this usually involves using a heartbeat mechanism to keep track of the server status and avoiding sending requests to the servers that are not responding.

Session-level Failover. Since an HTTP client can have a session that is maintained by the HTTP server, in session level failover, if one of the servers in the cluster goes down, then some other server in the cluster should be able to carry on with the sessions that were being handled by it, with minimal loss of continuity. In a typical clustering solution, this involves replicating the session data across the cluster (to one other machine in the cluster, at the least).

Load Balancing

The objective here is that the solution should distribute the load among the servers in the cluster to provide the best possible response time to the end user.

In a typical clustering solution, this involves use of a load distribution algorithm, like a simple round robin algorithm or more sophisticated algorithms, that distributes requests to the servers in the cluster by keeping track of the load and available resources on the servers.

Our Solution

Typical clustering solutions use a client-server paradigm to implement a distributed system as a solution, but they have limited scalability.

In the space paradigm that is used here, a request is fulfilled by having an object move from one machine to another, carrying with it the present state of execution and everything else needed, including the (byte)code, if needed, using an associative, distributed, shared memory.

Let us examine how the Jakarta Tomcat Servlet engine works. It uses a connector and a processor to receive and fulfill requests, as shown in Figure 2.

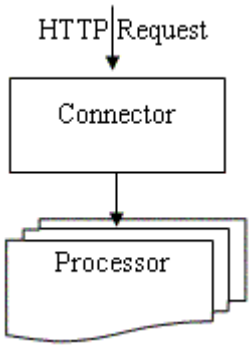

Figure 2. Architecture of a Stand-alone Tomcat Servlet Engine

The connectors abstract the details of receiving the request from different sources, while the processor abstracts the details of fulfilling the requests.

Figure 3 below shows the architecture of our proposed clustering solution.

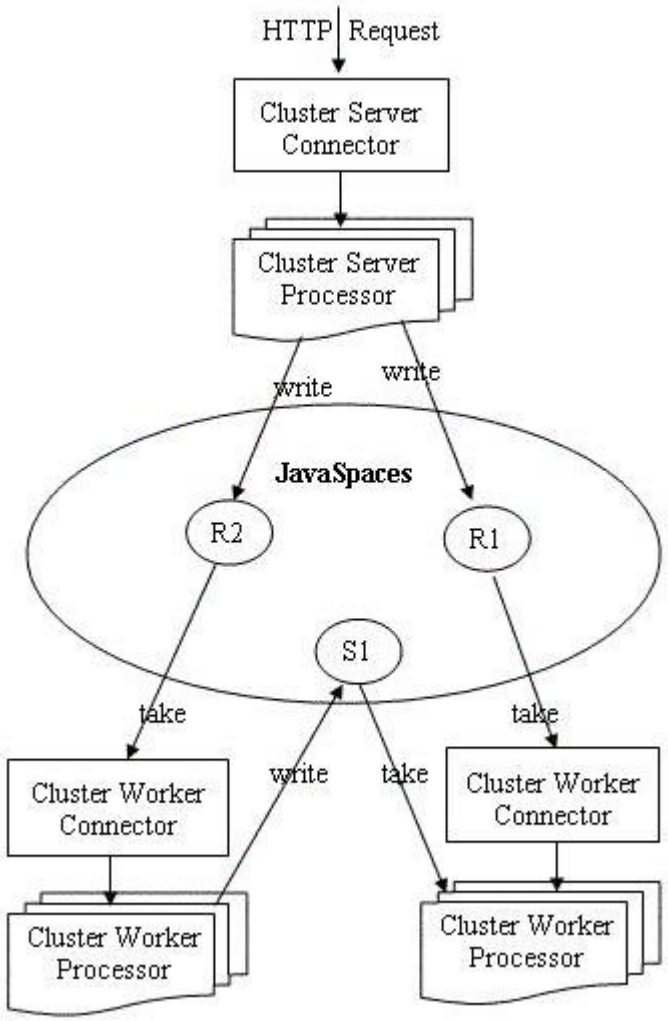

Figure 3. Architecture of the Clustered Tomcat Servlet Engine

The Cluster Server Connector receives the requests from the clients, and the Cluster Server Processor encapsulates the requests into `RequstEntry` objects and writes them into the JavaSpace. The Cluster Worker Connector then takes these requests from the JavaSpace and the Cluster Worker Processor then fulfills the request.

There can be multiple instances of these Cluster Servers and Cluster Workers, and they can be distributed across a number of machines.

The `RequestEntry` is defined as follows:

```
public class RequestEntry extends
      net.jini.entry.AbstractEntry {
  private static int ID = 0;
  public RequestEntry(){
    id = String.valueOf(ID++);
  }
  public String id;
  public RemoteSocketInputStream input;
  public RemoteOutputStream output;
}
```

The `id` field identifies a request. The input field is an object of type `RemoteSocketInputStream` that provides access to the input stream of a socket on a remote machine, and the output field is of type `RemoteOutputStream` that provides access to a remote output stream.

When a request is received, the socket's input and output streams are wrapped with the remote stream implementations that make them accessible from a remote machine. A request entry is created with these remote streams and written into the JavaSpace by the Cluster Server Processor as in the code below:

```
...

/**
* Process an incoming HTTP request on the Socket that has been assigned
* to this Processor. Any exceptions that occur during processing must be
* swallowed and dealt with.
*
* @param socket The socket on which we are connected to the client
*/


private void process(Socket socket) {
  RemoteSocketInputStream input = null;
  RemoteOutputStream output = null;
  try{
    //had to synchronize because of a threading problem with the TC class loader
    synchronized(this.getClass()){
      input = new RemoteSocketInputStreamImpl(socket, connector.getPort());
      output = new RemoteOutputStreamImpl(socket.getOutputStream());
    }
    log("socket address = " + input.getSocketAddress() +
```

```
    ", port " + input.getServerPort());
    RequestEntry entry = new RequestEntry();
    entry.input = input;
    entry.output = output;
    requestGenerator.write(entry);
  }catch(Exception ex){
  log("parse.request", ex);
  }
}

...
```

The Cluster Worker Connectors/Processors then take these entries and fulfill the requests, as shown in the code below:

```
...
public void run() {
  //process requests until we receive a shutdown signal
  while (!stopped) {
    log("waiting for entry in JavaSpace...");
    RequestEntry requestEntry = null;
    try{
      synchronized(this){
      ...
      requestEntry = (RequestEntry)requestReader.take();
      ...
      }
    }catch(Exception ex){
      log("internal error while getting request entry from JavaSpace", ex);
    }
    log("got an entry " + requestEntry);
    // process the request
    if (requestEntry != null)
      process(requestEntry);
  }
  ...
}
...
```

What Does Our Solution Provide?

This solution provides high scalability, because as the load increases, the number of Cluster Workers can be increased; also, these Cluster Workers can

be distributed across different machines. There can be multiple instances of the JavaSpace Service running on different machines, forming groups, and the Cluster Servers can decide to write to any of them, thus keeping the network traffic that is generated manageable.

The Cluster Servers and Workers use the Jini service discovery mechanism to find these JavaSpaces service implementations, so their locations do not need to be hard-coded. Thus, the solution is dynamically configurable. Use of the Jini leasing model allows additional resources to be added without bringing down the whole system. Similarly, existing resources can be taken down gracefully by not renewing the jini licenses and taking them out after the existing licenses expire.

Since the distribution of requests to the different servers in the cluster is by pull rather than by push, there is no question of requests getting directed to servers that have gone down, thus providing request-level fail-over.

In this solution, the session information is placed in the JavaSpaces and retrieved later using the Session ID provided by the HTTP client when needed. Thus, it provides session-level fail-over because even if the server that originally created the session dies, some other server in the cluster can carry on with the session. Since we're using the Jini leasing model, we can set a time after which the session entries in the JavaSpaces expire to the session timeout interval; this takes care of purging expired session data.

Since the distribution of load is based on pull, there is automatic load balancing, as the servers with more free resources are going to pull more work.

This design shifts the responsibilities of maintaining the session data to the JavaSpaces, and since there are different implementations of the JavaSpaces services that can provide persistence and also transactional capabilities, it becomes easier to provide persistent sessions if needed.

Related Series: Using Tomcat

Embedding Tomcat Into Java Applications -- James Goodwill shows how to create a Java application that manages an embedded version of the Tomcat JSP/servlet container.

Using SOAP with Tomcat -- The Apache SOAP Project is an open source Java implementation of SOAP. This article examines how you can can create and deploy SOAP services with Apache's RPC model.

Using Tomcat 4 Security Realms -- In part 4 of his Using Tomcat series, James Goodwill covers Tomcat 4, focusing on security realms using both memory and JDBC realms (with a MySQL database example).

Deploying Web Applications to Tomcat -- James Goodwill takes us through the web application deployment process for the Apache Tomcat Web server.

Installing and Configuring Tomcat -- James Goodwill covers the installation and configuration for the Tomcat Web Server.

Java Web Applications -- James Goodwill discusses the definition, directory structure, deployment descriptor, and packaging of a Tomcat web application.

Currently, we are using the space as a bag to hold the requests, but it is possible (with minor modification) to make it behave as a channel, as described in the article "Build and use distributed data structures in your JavaSpaces". Similarly, it is possible to implement a filtering service that will filter these requests; it is also possible to implement a service to prioritize these requests thus providing QoS capabilities.

Limitations of Our Solution

There can be more than one Cluster Server, and it makes no difference to how it all works, but HTTP clients typically use a URL to access Web resources, and the clients need to perceive the cluster as a single machine with one IP address (for server affinity), thus requiring a single Cluster Server.

Thus, this can be single point of failure, but this limitation can be overcome by using a hardware load balancer (even then, the load balancer itself is a single point of failure), in which case this solution can be used as the Web server proxy (as described in articles on J2EE Clustering by Abraham Kang), or by using other techniques to let more than one machine serve a single domain name. Another technique is described in the article "ONE-IP: Techniques for Hosting a Service on a Cluster of Machines".

Conclusions

This solution provides high scalability, high availability, and good load balancing capabilities that are comparable with any other software solution. It needs low maintenance because of the dynamic self-configuration capabilities, and has good promise for implementing QoS capabilities.

Download the source code for the solution.

Resources

- "Load Balancing Web Applications," by Vivek Viswanathan (OnJava.com)
- Articles on J2EE Clustering by Abraham Kang *(JavaWorld)*
- "*ONE-IP: Techniques for Hosting a Service on a Cluster of Machines*," by Damani et. al. *(Bell Laboratories, Lucent Technologies.)*
- The "Make Room for JavaSpaces" series by Eric Freeman and Susanne Hupfer *(JavaWorld)*

Shyam Kumar Doddavula works as a Technical Specialist for Infosys Technologies in SETLabs, the R&D division of the company.

Return to ONJava.com.