JAVAWORLD

NEWS

Use a distributed cache to cluster your Spring remoting services

Add automatic discovery and clustering to Spring remoting

By Mikhail Garber

JavaWorld | Oct 31, 2005 12:00 AM PT

As enterprise computing enters the brave new world of service-oriented architecture (SOA), it becomes more and more important to seek new ways to describe, publish, and discover your services. The Web services-based approach does not offer automatic service discovery and often is too complex. New, lightweight development frameworks warrant new, lightweight approaches to service publishing.

Over the past several years, the Spring Framework has emerged as the de facto standard for developing simple, flexible, easy-to-configure J2EE applications. At the heart of Spring lies the Inversion of Control (IoC) principle. According to IoC, an application must be developed as a set of simple JavaBeans (or plain-old Java objects—POJOs), with a a *lightweight IoC container* wiring them together and setting the dependencies.

In Spring's case, the container is configured via a set of *bean definitions*, typically captured in XML *context files*:

```
 <bean id="MyServiceBean" class="mypackage.MyServiceImpl">
   <property name="otherService" ref="OtherServiceBean"/>
</bean>
```

Then, when the client code needs to use `MyService`, as defined in this Spring context, you do something like:

```
 MyServiceInterface service = (MyServiceInterface)context.getBean("MyServiceBean");
service.doSomething();
```

In addition to IoC, Spring provides literally hundreds of other services, coding conveniences, and "hooks" into standard APIs that ease the development of a modern Java server-side application. Whether your application uses heavy-lifting J2EE APIs such as Enterprise JavaBeans (EJB), Java Message Service (JMS), or Java Management Extensions (JMX), or utilizes one of the popular Model-View-Controller frameworks for building a Web interface, Spring offers you something to simplify your development efforts.

As the Spring Framework matures, more and more people are using it as a foundation for their large-scale enterprise projects. Spring has passed the test of development scalability and can be used as sort of "component glue" to put together complex distributed systems.

Any nontrivial enterprise application combines many diverse components: gateways to legacy and enterprise resource planning systems, third-party systems, Web/presentation/persistence tiers, etc. It is not unusual for an e-commerce site that began as a simple Web application to eventually grow to contain hundreds of subapplications and subsystems, and face a situation where the complexity starts inhibiting further growth. Often the solution is to break the monolithic application into a few coarsely-grained services and release them on the network.

Whether your application was designed as an integration point for dispersed services or has morphed into one, the task of managing all distributed components and their configuration quickly becomes a time-consuming and expensive one. If your application components are developed using Spring, you can use Spring remoting to expose your Spring-managed beans to remote clients via a multitude of protocols. Using Spring, making your application distributed is as simple as making a few changes in your Spring context files.

The simplest (and most recommended) approach to Java-to-Java remoting in Spring is through HTTP remoting. For example, after registering your Spring dispatcher servlet in `web.xml`, the following context piece exposes `MyService` for public consumption:

```
 <bean name="/MyRemoteService" class="org.springframework.remoting.httpinvoker.HttpInv
okerServiceExporter">
  <property name="service" ref="MyServiceBean"/>
  <property name="serviceInterface" value="mypackage.MyServiceInterface"/>
</bean>
```

As you can see, the actual service is *injected* into this bean definition and thus made available for the remote calls.

On the client, the context definition reads:

```
 <bean id="MyServiceBean"
      class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
      <property name="serviceUrl"
         value="http://somehost:8080/webapp-context/some-mapping-for-spring-servlet/My
RemoteService" />
      <property name="serviceInterface"
         value="mypackage.MyServiceInterface" />
   </bean>
```

By the magic of Spring, the client-side code (obtain the service from the context and invoke its methods) doesn't change, and the remote method invocation occurs just as the local one did before.

In addition to HTTP remoting, Spring supports several other remoting protocols out of the box, including other HTTP-based solutions (Web services, Hessian, and Burlap) and heavier ones like remote method invocation (RMI).

Configure and deploy URL-based remoting services

Deploying your services via HTTP-based remoting has several distinct advantages, one of which is that, compared with straight RMI or EJB-based solutions, you have far fewer configuration issues to worry about. Anyone who has tried to work through a nontrivial JNDI (Java Native and Directory Interface) configuration (several load-balanced or clustered J2EE containers from different vendors or even different versions of the same container) can attest to that.

If you base your distributed components on Spring remoting, defining a service on your network is simple. All you need to know is the service URL pointing to the server, port, Web application, context path, and name of the Spring bean implementing this service.

URLs are plain-text strings, and plain text is your friend. At the same time, defining a service via a URL makes the definition somewhat brittle. All the individual portions of the URL listed in the previous paragraph are subject to change, and change they will. Network topography (and network administrators) change, load-balanced server farms replace servers, Web applications deploy onto different containers under different names, holes are punched and closed in inter-network firewalls, and so on.

In addition, those brittle URLs must be stored in Spring context files on every client that could possibly access the service. When they change, all the clients must be updated. And one more thing: As your newly-forged service progresses from development to staging to production, the URL pointing to the service must change to reflect the environment the service is in.

Finally we arrive at the problem definition: Spring's ability to easily expose individual Spring-managed beans as remotely-accessible services is great. It would be even better if all we needed to define (or access) a service was the *service name*, with all the details about service location hidden

from the clients.

Cache service descriptions for auto-discovery and failover

The obvious solution to this problem would employ some kind of naming service to provide dynamic, real-time (or almost real-time) resolution of service name to service location(s). Indeed, I once built such a system using the JmDNS library to register Spring remoting services in the Zeroconf namespace (Zero Configuration Networking, technology also known as Apple Rendezvous).

The problem with the DNS-based approach is that updates to the service definitions are never real-time or transactional. A failed server still appears in the service list until all kinds of timeouts and "keep-alive" games are played. What we need is the ability to quickly publish and alter the lists of URLs implementing our services and make those changes happen simultaneously (read: transactionally) across our entire network.

The systems that satisfy these requirements are available. They are various implementations of a distributed cache. The easiest way to visualize a cache for a Java programmer is to think of it as an implementation of the `java.util.Map` interface. You can *put* something in there using a key, and then you can *get* something out using the same key later. A distributed cache ensures that the same key-value mapping will exist in all the copies of the same `Map` on every server participating in the cache and will update the caches everywhere in a lockstep.

A good implementation of a distributed cache solves our problem. We associate a service name with one or more URLs pointing to the place(s) on the network where this service is implemented. Then, we store the name=(list of URLS) associations in a distributed cache and update them accordingly as the network situation changes (servers come online and are removed, servers crash, etc.). The clients to our services participate in the distributed cache as well and, as such, always have access to the current information about the individual service implementations' locations.

As an added bonus, we can introduce a simple load-balancing/failover solution in this scenario. If a client knows that a certain service is associated with several service URLs, it can pick one of them at random and provide crude but effective load-balancing across the several servers serving those URLs. And, if a remote call fails, a client can simply mark that URL as "bad" and pick the next one, thus providing failover as well. Because the list of service URLs is stored in the distributed cache, the fact that Server A went bad is communicated to the other clients as well.

Distributed caches find use in conventional J2EE applications that provide the backbone for server clustering. For example, if you have a distributed, clustered Web application, a distributed cache will provide session replication among your cluster's members. Though highly reliable, J2EE clustering is a serious bottleneck. Session data change quickly, and the overhead of updating all the cluster members and failing over in case of failure is great. Clustered Web applications with session replication are typically several times less scalable than share-nothing load-balancer-based solutions.

Distributed caching works for our scenario due to the small amount of data being cached. Instead of

thousands of session objects typical for distributed session replication, we have only a small list of services and the URLs implementing them. In addition, updates to our list happen infrequently. A distributed cache with such a small list may scale well to numerous member servers and clients.

For the rest of this article, let's look at the real-life implementation of our "service description caching algorithm."

Use Spring and JBoss Cache for service description caching

JBoss Application Server is probably the most successful (and the most controversial) open source J2EE project today. Love it or hate it, JBoss Application Server occupies a well-deserved spot on the list of top deployed servers, and its modular nature makes it very developer-friendly.

The JBoss distribution packs many ready-to-go services. One of interest to us is JBoss Cache. This cache implementation provides high-performance caching of arbitrary Java objects both locally and across the network. JBoss Cache has many configuration options and features, and I encourage you to learn more about it to see how it may fit into your next project.

The features that make JBoss Cache attractive for our project are:

- It provides high-quality, transactional replication of Java objects
- It can run as part of JBoss server or standalone
- It is already available "inside" JBoss as an MBean (managed bean)
- It can use either UDP multicast or "normal" TCP connections

The network foundation for JBoss Cache is JGroups library. JGroups provides network communication between cluster members and can work over either UDP multicast (for dynamic auto-discovery of cache members) or over TCP/IP (for working off a fixed list of server names/addresses).

For this article, I show how to use JBoss Cache to store the definitions of our services and provide dynamic, automatic service discovery.

Note: See Resources to download a zipped file containing an Eclipse project for a Web application that exposes a service via Spring remoting and uses JBoss Cache to share the service descriptions with a client application (set of JUnit tests). All of the code discussed below can be found there.

To begin, we introduce a custom class, `AutoDiscoveredServiceExporter` that extends the Spring standard `HttpInvokerServiceExporter` to expose our `TestService` for remoting:

```
 <bean name="/TestService" class="app.service.AutoDiscoveredServiceExporter">
  <property name="service" ref="TestService"/>
  <property name="serviceInterface" value="app.service.TestServiceInterface"/>
</bean>
```

There is really nothing worth mentioning in this class. We basically use it to mark the Spring remoting services as exposed in our special way.

Next, the server-side cache configuration. JBoss already comes with a cache implementation, and we can use the Spring built-in JMX proxy to bring the cache into the Spring context:

```
 <bean id="CustomTreeCacheMBean" class="org.springframework.jmx.access.MBeanProxyFactoryBean">
    <property name="objectName">
      <value>jboss.cache:service=CustomTreeCache</value>
    </property>
    <property name="proxyInterface">
      <value>org.jboss.cache.TreeCacheMBean</value>
    </property>
</bean>
```

This creates a `CustomTreeCacheMBean` in the server-side Spring context. Through the magic of auto-proxying, this bean implements the methods in the `org.jboss.cache.TreeCacheMBean` interface. For this to deploy on the JBoss server, just drop the provided custom-cache-service.xml file into your server's deploy directory.

To simplify our code, we introduce a simple `CacheServiceInterface`: