FERMILAB-CONF-10-549-CD

Horizontally scaling dCache SRM with the Terracotta platform

T Perelmutov[1], M Crawford, A Moibenko and G Oleynik

Fermi National Accelerator Laboratory, MS-120, PO Box 500, Batavia, Illinois 60510 USA

E-mail: timurp@gmail.com, crawford@fnal.gov

Abstract. The dCache disk caching file system has been chosen by a majority of LHC experiments' Tier 1 centers for their data storage needs. It is also deployed at many Tier 2 centers. The Storage Resource Manager (SRM) is a standardized grid storage interface and a single point of remote entry into dCache, and hence is a critical component. SRM must scale to increasing transaction rates and remain resilient against changing usage patterns. The initial implementation of the SRM service in dCache suffered from an inability to support clustered deployment, and its performance was limited by the hardware of a single node. Using the Terracotta platform[1], we added the ability to horizontally scale the dCache SRM service to run on multiple nodes in a cluster configuration, coupled with network load balancing. This gives site administrators the ability to increase the performance and reliability of SRM service to face the ever-increasing requirements of LHC data handling. In this paper we will describe the previous limitations of the architecture SRM server and how the Terracotta platform allowed us to readily convert single node service into a highly scalable clustered application.

1. Storage Resource Manager

The Storage Resource Manager (SRM)[2] is a web service protocol for managing a hierarchical Mass Storage System on the grid. "Storage Resource Managers (SRMs), named after their web services protocol, provide the technology needed to manage the rapidly growing distributed data volumes, as a result of faster and larger computational facilities. SRMs are Grid storage services providing interfaces to storage resources, as well as advanced functionality such as dynamic space allocation and file management on shared storage systems."[2] SRM was selected by the Worldwide LHC Computing Grid (WLCG) as a management protocol to data storage systems that are components of the WLCG Data Grid. DCache has been chosen by a majority of LHC Experiments' Tier 1 centers, and stores more data that any other disk storage used by LHC Experiments. Dcache SRM interface has been implemented by Fermilab and was funded by DOE and US CMS. Work on achieving Horizontal Scalability in dCache SRM server was mostly funded by US CMS.

2. SRM performance issues

DCache SRM is a single point of remote entry into the system, and its performance is critical to a successful operations of dCache based LHC Data Storage Systems. Experiments reported reaching or nearing the performance limits of dCache SRM, and a review[3] was conducted.

[1] Present address: Navteq, 425 West Randolph Street, Chicago, Illinois 60606 USA

One of the recommendations from that review was "The plan to load balance SRM horizontally should be pursued at a high priority," another one was "SRM (and dCache) should evaluate using Terracotta in front of distributed database accesses."

We evaluated Terracotta's functionality and found that it could be used to develop a multi-node load balanced dCache SRM Server. Implementation of simple prototypes demonstrated that Terracotta works as advertised. Comparing with other technologies, using Terracotta appeared to require fewer changes in dCache SRM code. The number of positive experience reports on the web was another factor that led to the decision to pursue the integration of SRM with Terracotta in order to achieve horizontal scalability and load balance the SRM service.

3. Terracotta platform description

> Terracotta is open source infrastructure software that makes it inexpensive and easy to scale a Java application to as many computers as needed, without the usual custom application code and databases used to share data in a cluster.[1]

A Java application that relies on the Terracotta platform can store java objects of designated classes in Networked Attached Memory managed by Terracotta servers. The application does not have to use any of the Terracotta classes or interfaces. Certain configuration parameters need to be passed to the Java Virtual Machine (JVM), to allow the Terracotta platform to take over the memory management of the specified classes. Though a simple configuration file certain fields are declared "Terracotta Roots," making those fields "super-static." After that declaration, the objects referenced from these fields and the entire tree of objects reachable from these roots becomes managed by Terracotta and stored in Network Attached Memory. If there are several JVMs which are members of the same Terracotta cluster, they all see the same object trees reachable from each Terracotta Root field. For example, if the Terracotta root is a Map, than an object put in the map by code in one JVM can be retrieved by code running in a different JVM. Also the standard java mechanisms for inter-thread communication and synchronization now become clustered objects and allow communication and synchronization between threads in different JVMs. One more feature of Terracotta worth mention is the support for distributed methods. Methods that are declared distributed, when invoked in one JVM, will be invoked in all JVMs of the cluster.

4. SRM-Terracotta integration

The simplicity of the Terracotta model might suggest that dCache SRM could be run in clustered mode on top of Terracotta without modification. But experiments showed that the opposite is true: SRM code required significant refactoring and updates as described below in order to benefit from Terracotta. Let us first give a few details of the functions of a distributed SRM that would require sharing state among multiple instances of the server. In dCache SRM there are the following functions that require creation and maintenance of state objects on the server:

`srmPrepareToGet`	`srmPrepareToPut`	`srmBringOnline`
`srmCopy`	`srmLs`	`srmReserveSpace`

The partial class diagram in figure 1 shows SRM Classes that are created on the dCache Server as the result of the invocation of the above functions. We will further refer to the the instances of these classes as just jobs, since Job is the superclass of all such objects.

Once a job is created, the reply to the SRM status functions can be computed from the job instance. In order to reply to the status inquiry from any server in a cluster, each of the servers must be able to obtain a copy of the up-to-date Job objects. Some client functions, such as `srmPutDone`, `srmReleaseFiles`, `srmAbortRequest` and others, can affect the state of the SRM request in the server through a call to job.setState(...). Other notable cases in which a client call leads to a change of state are the `get{Get,Put,Ls,BringOnline}RequestStatus` calls against

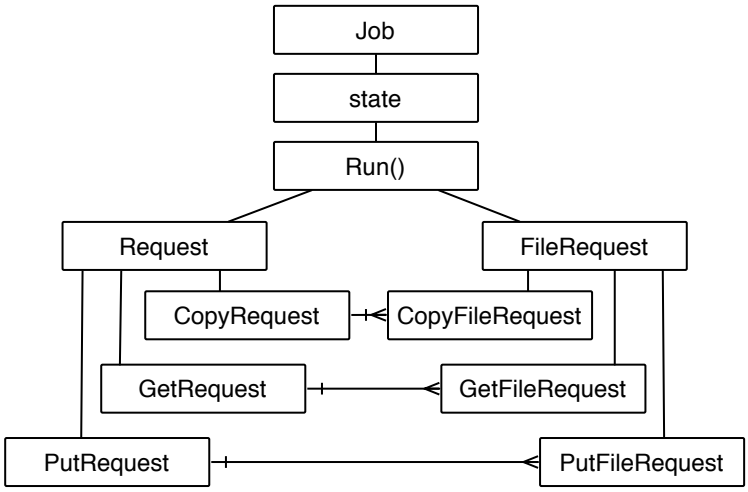

Figure 1. SRM requests class diagram

a job in the `ReadyQueued` state , which may lead to a transfer from `ReadyQueued` to `Ready` state, subject to the availability of the limited number of "ready slots." So the first order of business was to create a mechanism that would allow the sharing of instances of Job objects among the SRM servers, and the second task was to propagate all requests to change the state of a job to the SRM server that "owns" the particular job and can act on such requests.

4.1. Definition of Terracotta "roots" and "instrumented classes"
We defined a class SharedMemoryCache, which contains a HashMap that became the distributed storage for the Jobs. Each existing job is identified by a server-asigned unique id that is included in all SRM web service requests concerning the job. These ids were used as keys, and jobs themselves as values stored in the HashMap. SRM code was modified to store each job in the static Job.sharedMemoryCache field of type SharedMemoryCache when it is created and to remove the job from the SharedMemoryCache when its execution is completed. The last step to making the jobs shared was to let Terracotta know that the static field Job.sharedMemoryCache is a Terracotta Root. This was done by putting the following XML code in the Terracotta configuration file:

```
<roots>
  <root>
    <field-name>org.dcache.srm.scheduler.Job.sharedMemoryCache</field-name>
  </root>
</roots>
```

Furthermore Terracotta requires that all the types of all the objects that may possibly become distributed be declared in the Terracotta configuration as "instrumented classes." Here is an example of the XML added to the Terracotta configuration for this purpose:

```
<instrumented-classes>
  <include>
    <class-expression>org.dcache.srm.request..*</class-expression>
    <honor-transient>true</honor-transient>
  </include>
  ...
```

```
</instrumented-classes>
```

4.2. Identification and isolation of shared state
In the dCache SRM that existed prior to this project, jobs and job dependents used to have direct references to many objects that supported the execution of the job, but which could not be distributed to other nodes in the SRM cluster. Such objects included those that encapsulate local configuration, instances of local server credentials, connections to databases, references to the dCache-specific objects used for dCache internal communication. SRM jobs had to be modified so that they did not directly reference these objects, so that when replicated by Terracotta, these objects and those they reference directly and indirectly are not "dragged along" with them into other instances of SRM. We accomplished that by extensively using the Factory Method and Abstract Factory Design Patterns and never storing direct references in our objects. These changes also made SRM Code easier to read and maintain.

4.3. Protecting shared state with locks
Once an object is handed over to Terracotta for management by being stored in the Object tree reachable from the "Terracotta Root," all read and write access to the mutable fields of that object need to be protected with lock objects that are themselves managed by Terracotta (making them distributed locks). Any access to an unprotected field results in a runtime exception. This initially led to a large number of failures in execution of the SRM code with Terracotta, in completely unexpected places. We chose to use Reentrant Read Write Locks from java.util.concurrent.locks package. When run under Terracotta, these locks are replaced by Terracotta's own locks. We initially used the Read Locks to protect reads and Write Locks to protect writes but discovered that the Terracotta version of the locks were not upgradable. (Code protected with a read lock can not be further in its execution be upgraded to be protected with write locks.) This led us to just use Write Locks everywhere. Since distributed lock and unlock operations are expensive and in order to minimize the use of locks, we identified and made immutable (java "final") the fields that really do not need changes to their values.

4.4. Defining distributed methods in SRM
In distributed SRM the node that receives a call resulting in a creation of a job object is defined as the owner node of that job, and this owner node will use its local SRM Scheduler for execution of the job. This is the node that should ultimately receive the request to change the state of the job and this is the SRM that will do the proper accounting of resources used for the execution of the job. Therefore if another node than the owner receives a call that results in a change of state, it must be able to propagate the call to the owner node. We accomplished this sort of messaging though the use of Terracotta "distributed" methods. Any method that is declared distributed is invoked simultaneously on each instance of the service in the cluster where a copy of the affected object is present in the heap. The only changes that we had to make to the methods themselves was to ensure that accesses to the fields manipulated in the methods are protected by the locks. We declare methods as Terracotta distributed by including the following code in the configuration:

```
<distributed-methods>
  <!-- An AspectWerkz-compatible method specification
       expression denoting which method(s) to distribute. -->
  <!-- An optional attribute run-on-all-nodes (default value "true")
       can be set to false to execute distributed only on those nodes
       that already have a reference to the object on which the method
       is called -->
```

```
    <method-expression run-on-all-nodes="false">
      void org.dcache.srm.scheduler.Job.tryToReady()
    </method-expression>
  </distributed-methods>
```

4.5. Startup script

The Terracotta distribution includes a tool `dso-env.sh`, which sets the `TC_JAVA_OPTS` environment variable, the value of which needs to be included as one of the arguments of the java binary distribution. `Dso-env.sh` relies on the values of the `TC_INSTALL_DIR` and `TC_CONFIG_PATH` variables. So modification of the dCache startup scripts was a trivial task of making sure that the `TC_INSTALL_DIR` and `TC_CONFIG_PATH` are defined if Terracotta usage is enabled, and then running `dso-env.sh` and inserting the value of `TC_JAVA_OPTS` in the command line starting dCache.

5. Conclusion

We set out to create a distributed SRM server powered by Terracotta. The Terracotta platform documentation had a clear recipe for converting a plain java application into a clustered one. The technology was simple and mostly transparent to java application and provided the functionality we needed to make distributed dCache SRM a new deployment option. With this new feature, it is possible to deploy a number of the dCache SRM servers, with each one running on a different node in a singe dCache instance, and, using a network load balancer, it is possible to make all these servers appear as a single service endpoint. In addition to potentially increased performance, this architecture also provides for greater availability, since the dCache SRM service will continue to perform even if one of the SRM server nodes fails.

For measurements of Terracotta clustering's effect on the performance of the dCache SRM server, please see the accompanying paper from the BNL dCache group. Here, we will just say that the gains from the multiple parallel SRM server nodes were almost entirely offset by synchronization overheads. At least two avenues for improvement are under study. The first is to use a ConcurrentHashMap instead of a HashMap for the registry of request ids and job objects. The second approach is to have the "wrong server" receiving an operation concerning an existing job discover the job's owner node through an ad-hoc local multicast query or through a predefined partitioning of the request id space.

References

[1] Terracotta I 2010 The terracotta scalability platform URL `http://www.terracotta.org/platform/`

[2] Sim A and Shoshani A (eds) 2008 The storage resource manager interface specification version 2.2 Tech. Rep. GDF.129 Open Grid Forum URL `http://www.ogf.org/documents/GFD.129.pdf`

[3] Oleynik G, Crawford M, Behrmann G, Dumitrescu C, Gysin S, Levshina T, Salgado P and Perelmutov T 2009 Srm scalability/performance review URL `http://srm.fnal.gov/srm/review/docs/SRM-review-report.pdf`