Software Architecture: MSE Project

September 3, 2010

Prepared by Doug Smith
Version 0.2

Table of Contents

Table of Figures

Revision History

Version	Date	Changes
0.1	1/16/2009	First draft.
0.2	7/20/2009	Updates based on inspection feedback.
0.3	9/3/2010	Updated to incorporate changes to the architecture based on scale and performance testing

Introduction

This document outlines the software architecture for my MSE project. This project involves building a small process execution engine that uses process description metadata to define the execution of processes. The system is to be architected in such as way as to parallel the architecture of an enterprise workflow system, to use a replicated cache, and to tie these two constraints together such that the configuration of the replicated cache and the scale and performance characteristics of the small process engine can be used to draw conclusions about how to configure the enterprise workflow system.

For details on the process engine being designed and built for this MSE project, refer to the vision document [1].

The outline of this document follows the outline of the software architecture document described on the "Coding the Architecture" site, written by Simon Brown [2]. The purpose of this document, articulated by Simon Brown, is to provide:

1. An outline description of the software architecture, including major software components and their interactions.
2. A common understanding of the architectural principles used during design and implementation.
3. A description of the hardware and software platforms on which the system is built and deployed.
4. Explicit justification of how the architecture meets the non-functional requirements.

Two Variants of the Architecture Produced

The software architecture for MSE projects is defined and validated during the elaboration part of the project, with a variant of the Rational Unified Process defining the software development project methodology framing the activities and deliverables.

The primary objectives of the elaboration phase are to "define, validate, and baseline the architecture".[7]. Given this project is about producing a scalable architecture, the elaboration phase included architecture validation: actually running some load against a working system to demonstrate scalability characteristics.

After producing the first working system and scale testing it, I ran into a number of problems that invalidated the architecture:

- The initial baseline used software (JBoss Cache 1.4.1 SP12) that did not work due to bugs when changing the cluster communication protocol from multicast to a tcp using agossip protocol to establish cluster membership and cumminication links.
- Updating the ORM software (Hibernate) and caching software (JBoss Cache) worked in the EC2 environment, but showed completely unexpected results when doing scale testing. After enabling trace logging, it was revealed that the default behavior of the JBoss replicated cache software is to silently ignore replication errors if data or invalidation replication fails, and that replication had been failing during the testing I had done.
- After these failures, I was able to used the Hazelcast IMDG as the second level Hibernate cache and demonstrate cluster member data backups and entity and collection caching worked as expected. However, as documented further on the system did not exhibit the desired scalability characteristics.

- In response the scale characterization based on Hazelcast being used as a second level cache for Hibernate, I created a second architecture that uses Hazelcast as an In-Memory Data Grid (IMDG): this is the architecture that serves as the baseline for this project, and is described in this document along with the initial attempt at the architecture.

A Word on Modeling

"Essentially, all models are wrong, but some are useful"
--George Box

The architecture description in the document makes use of several models to describe the different views of the system. The level of detail in the models for this document is meant to convey the structure and mechanisms of the system design, but not fully specify the system to the extent it could be handed to a development team for implementation without additional design work.

As an example, the collaboration of objects to realize an architecturally significant use is shown, but the actual data structures passed in the individual method calls will be detailed in the construction phase of the project. The actual data being passed can be inferred from the database design and the formal specification document [3].

Part 1 – Initial Architecture Design

Functional View

The Vision document describes the use cases associated with the process execution engine being built for this project. The uses cases can be classified into two classes – one class is related to defining and retrieving process description metadata, and the other class is related to creating and updating process instances.

The first class of use cases, those related to process metadata, contribute two architecturally significant use cases: reads of process metadata lists data by clients, and updates to process metadata.

In an enterprise system, a client may read lists that contain hundreds or thousands of field definitions, for example. This becomes architecturally significant as the right caching solution can improve the responsiveness of the system to these requests, and avoid tying up resources with servicing lists requests which could potentially delay servicing of higher value requests, i.e. work processing. Also, replication of lists via query replication can dramatically affect the performance of the replicated cache, as it can result in the need to replicate large amounts of data to each cache node.

Updates to process metadata, while occurring relatively infrequently, are architecturally significant as they force an update or invalidation to be replicated to all cache instances.

The second class of use cases is in general architecturally significant, as they are high volume transactions with service level agreements around producing a sub-second response for thousands of concurrent users. These transactions combine reads of metadata to validate the input data, and to determine the new state of a process as determined by the transaction input data and the process metadata.

For the purposes of this document, three uses cases described in the Vision document cover the architecturally significant functionality:

- Retrieve Property List
- Update Property Description
- Execute Task

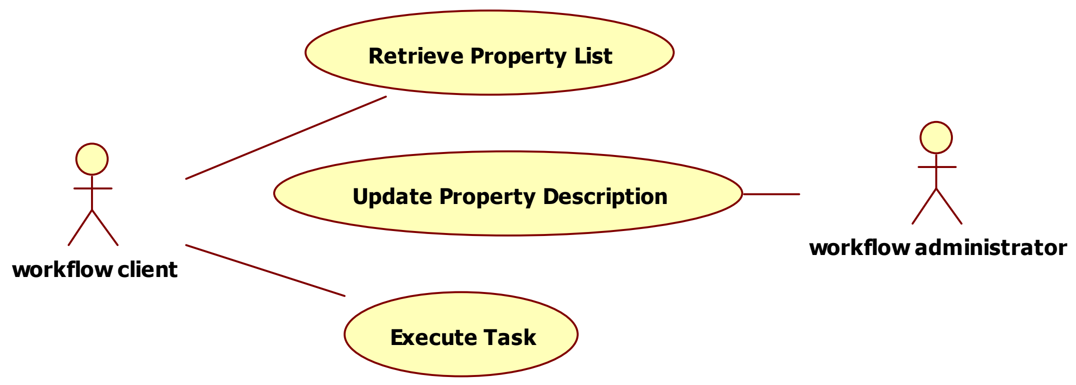


Figure 1 Architecturally Significant Use Cases

Note there are some significant operational use cases that will need to be tested; for example, the ability to start additional servers that join the cluster while the system is under load. These are not detail here as they do not have to be implemented in the construction of the software, but the configuration and characteristics of the system must be such that the system quality attributes operational use cases map to are satisfied.

Non-functional View

The vision document outlined the following system quality attributes:

- QA1 - The solution may not degrade average response time or throughput for transactions involving cached data. This requirement will be evaluated relative to the baseline performance characteristics of the system with caching disabled.
- QA2 - Functional correctness must be maintained when caching is enabled.
- QA3 - Application availability must not be compromised by the caching solution. The solution must provide the ability to survive failure of cluster members without compromising application availability or system correctness.
- QA4 - The solution shall provide a role-based security model to constrain access to the functions and data associated with the system based on roles.
- QA5 – the solution shall provide basic authentication and identity management, where users are required to authenticate prior to accessing the system, and user identities associated with one or more roles.
- QA6 – the solution must be scalable to 16 JVMs evenly distributed among 4 physically distinct servers, and reduce database requests for workflow metadata by at least 50%.
- QA7 - data access must be consistent across all JVMs serving application requests. It is unacceptable for results produced by the application to differ based on the server or JVM servicing the application request. In other words the view of the data across all cluster members must be consistent.
- QA8 - the solution must provide the ability to add and remove cluster members as needed without causing errors.
- QA9 – use of computing resources in the persistence tier will be reduced when the caching solution is enabled.

Architectural Principles

The following architectural principles should be noted:

- *Stateless Servers* – To simplify load balancing, failover, and provisioning, the implementation on the web servers shall be stateless; no state will be maintained between service method invocations in the server code. Any persistent state shall be maintained in the database.

Architectural Constraints

Given the system to be built is a model of a larger enterprise system, to support external validity of experiments with distributed cache architecture and configuration the design of the system must be closely aligned with the target system, using common software components and architectural patterns where possible.

Thus, an architectural constraint is the use of the software stack shown below. The top layer represents data access objects written to provide application data access. Hibernate is used as the ORM technology, accessed through the Spring Framework's ORM APIs.

Hibernate defines a pluggable cache provider interface, allowing cache providers to plug in a cache by providing implementations of the Cache and CacheProvider interfaces.

The Cache interface defines a set of operations for interacting with the cache to access data in the cache, add and remove data to/from the cache, lock and unlock items in the cache, and so on. The CacheProvider interface provides an interface to instantiate a cache and to start and stop it.

The Cache interface defines the base capabilities required for caching. Cache strategy classes define algorithms based on the Cache interface for caching solutions with different properties. For a replicated cache that uses transactions, Hibernate defines a class named TransactionalCache that provides the interaction with the Cache implementation that supports transactionally consistent caches. Currently, the TreeCache implementation is the only transactionally consistent cache supported directly via the Hibernate release. Note that configuration parameters associated with a cache provider determines the strategy class used with the cache, with the strategy class implementing the canonical Gang of Four Strategy pattern.

For using the JBoss cache (previously known as TreeCache), the TreeCache and TreeCacheProvider interface implementations provided with the Hibernate release are used. These in turn use the JBoss cache API to as the cache provider implementation.

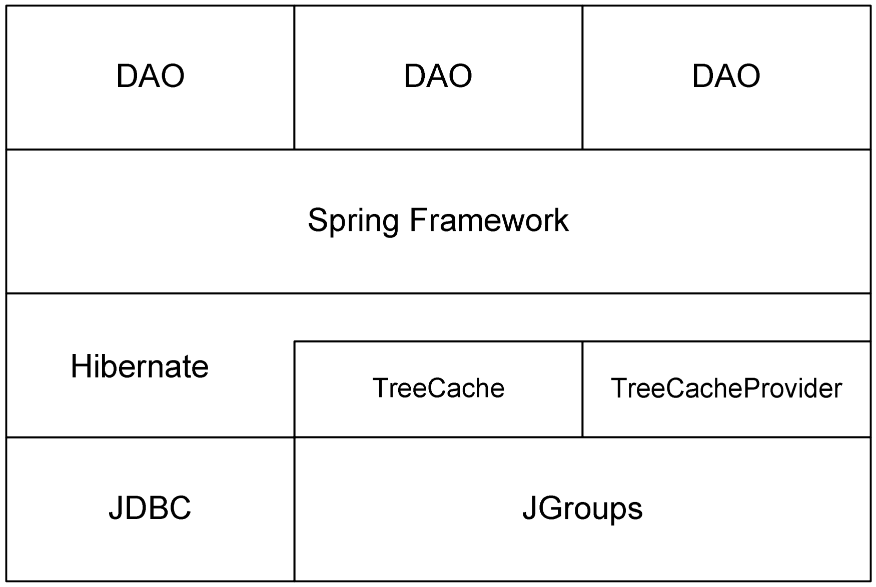


Figure 2 Software Stack

For cache replication, JGroups is used for communication between cluster members in a distributed JBoss cache configuration. JGroups is a reliable multicast protocol implementation.

Process View

The web services being built for this project are simple request response processes – no concurrency is needed in their implementations, and all thread management is encapsulated within the web services container hosting the application services.

Web services that modify cached data are associated with a JTA transaction, with the database and the replicated cache as the resource managers participating in the transaction. Modification of cached resources is handled via multicast communication among cache members using JGroups as the underlying protocol.

For the multicast communication associated with cache replication, the JGroups implementation uses concurrency for parallel processing of messages received from different senders, for dealing with out of band messages, and for protocol timers.

The following figure (taken from the JGroups document [4]) shows the use of threads in JGroups.

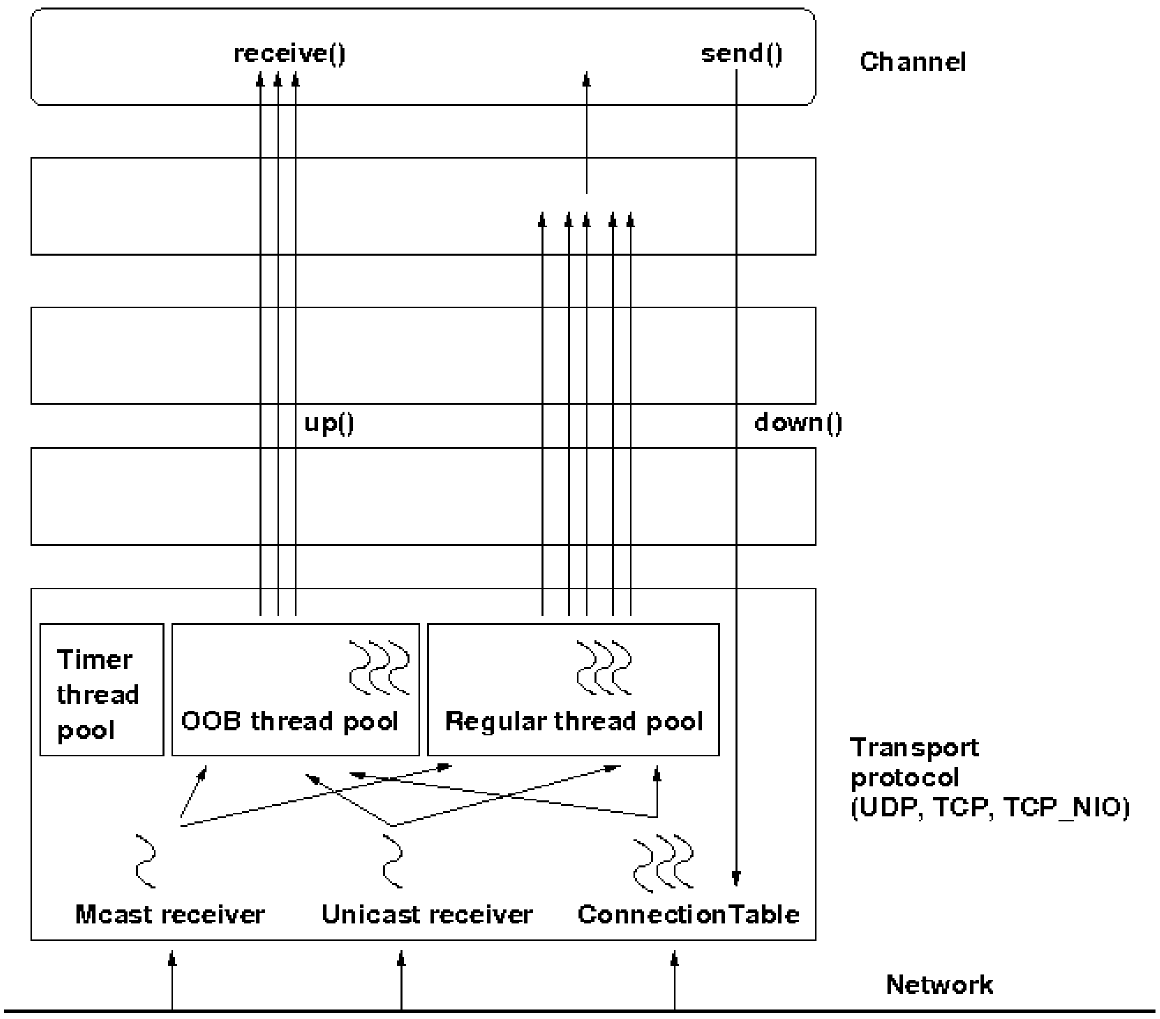


Figure 3 JGroups Thread Architecture

The transport layer use threads from the various pools for processing message, passing the message up the protocol stack using a thread allocated from a thread pool.

The thread pool size can be configured in the JGroups configuration, and thus will need to be considered when tuning the performance of the solution.

Logical View

The following class diagram, taken from the formal specification, shows the classes that represent the fundamental abstractions of the problem domain and the relationships between them.

The classes can be divided into process definition classes (Connectable, GatewayDefinition, ActivityDefinition, ProcessDefinition, Swimlane, PropertyDefinition, Role, ConnectorDefinition) and process execution classes (ProcessInstance, ActivityInstance, PropertyValue), with the User class representing workflow participant identity.

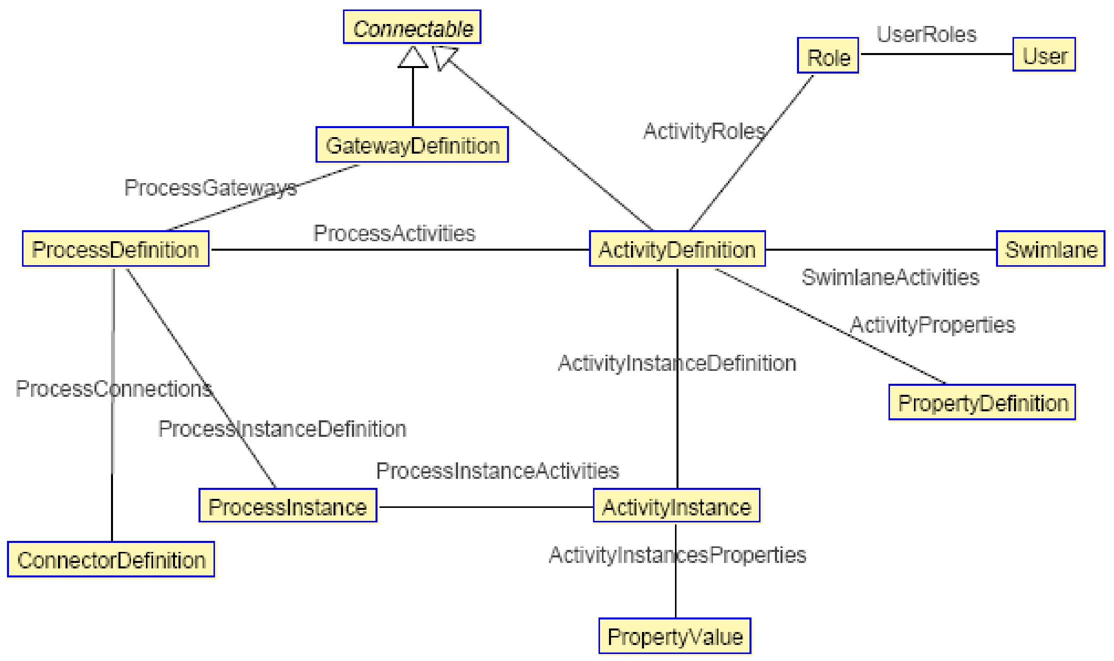


Figure 4 Logical Class Model

Process definition classes represent the low volatility metadata that will be cached for use by the process execution services.

The Design View section of this document shows the mapping of the logical entities to classes, the classes that represent the behavioral aspects of the system, and the service interface to the system.

Interface View

The interface to the system will be via WSDL-described web services, conforming to the WS-I 1.x Basic Interoperability Profile. The service interface contracts will be fully specified during the construction phase of the project.

Technology Selection

The technology selection for this project has two primary drivers. The first driver is to mirror as closely as possible the target technology stack associated with the enterprise application that represents the target environment from any learning or solutions developed as part of this project.

The second driver is to pick a technology stack that can be readily packaged and deployed to the test infrastructure, specifically the KSU Beocat cluster [5].

Design View

This view of the architecture shows the information, behavioral, and service interface view of the domain model and the interaction of the classes to realize the architecturally significant use cases.

Package Structure

The following figure shows a package view of the system. The spring and hibernate packages represent classes from the Spring and Hibernate frameworks, respectively. The service, persistence, and domain packages represent the classes that implement the system's functionality.

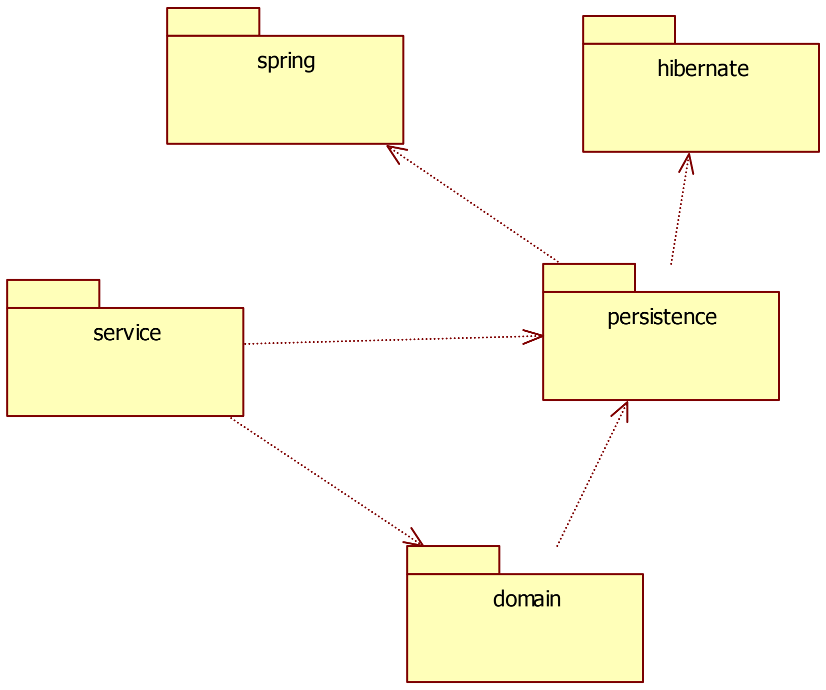


Figure 5 Package Diagram

The web services are implemented in the service package. These classes represent the service interface to the system, handling the service inputs to orchestrate interactions with the domain and persistence classes to realize the use cases.

The domain package represents the information view of the system, with the logical entities in the conceptual model mapped to classes in the domain package.

The persistence package contains Data Access Objects (DAOs) the provide persistence services for the entities in the system. It is in the persistence package that the use of the distributed cache is injected into the system via Hibernate configuration.

Structural View

The service classes for the architecturally significant use cases are the Activity and PropertyDefinition classes.

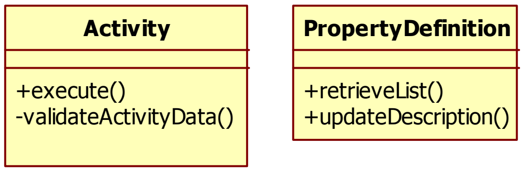


Figure 6 Service Classes
As the classes represent services, they do not maintain state, and thus do not have any member variables.

The following diagram shows the classes in the domain package.

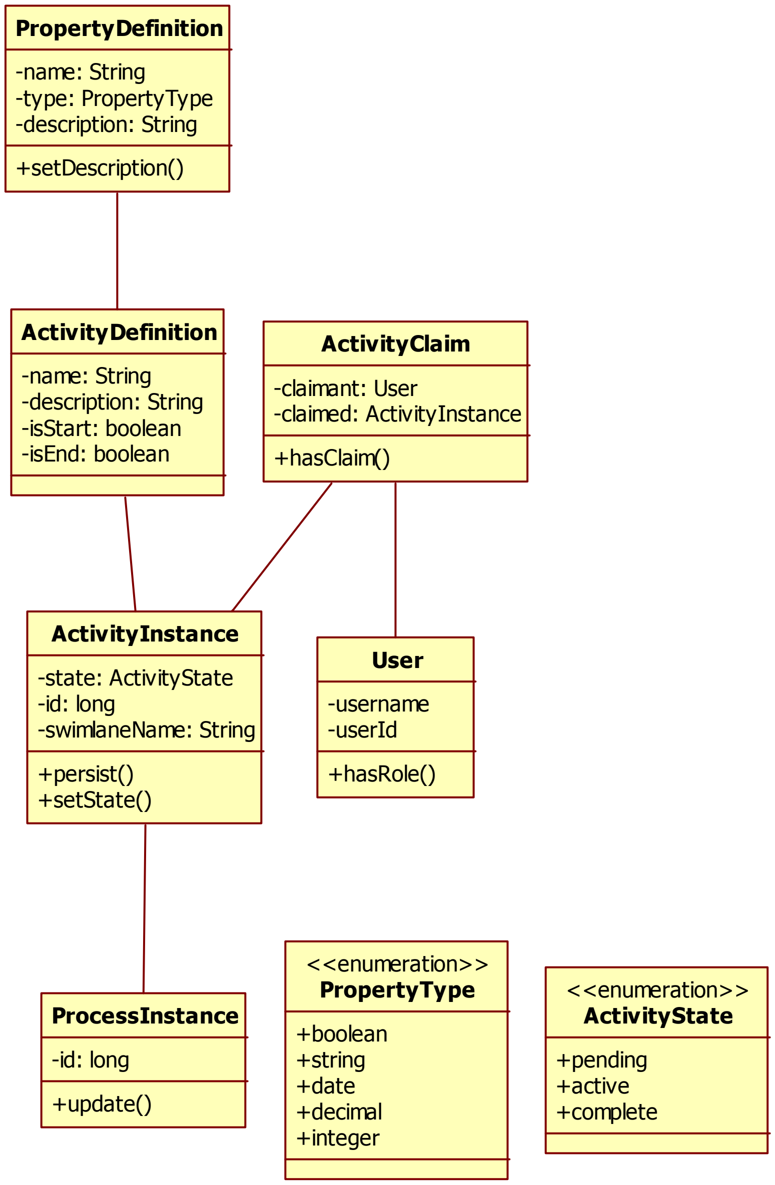


Figure 7 Domain Classes

These classes align with the conceptual model in the formal specification and in the logical design section of this document from an information modeling perspective. Some of the behavior has been moved to the service package.

Finally, for accessing and persisting data, the domain package contains Data Access Object classes:

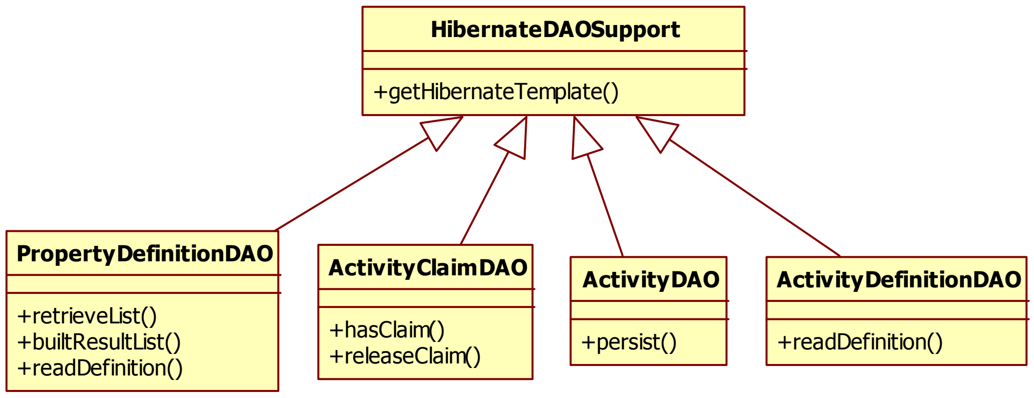


Figure 8 Persistence Classes

Note all the DAOs inhereit from the HibernateDAOSupport class, which provides a convenient way to inject a Spring HibernateTempate instance into the DAO to enable the use of the Spring persistence API for data access.

Behavior View

The following sequence diagrams show how the architecturally significant use cases are realized.

Retrieve Property List

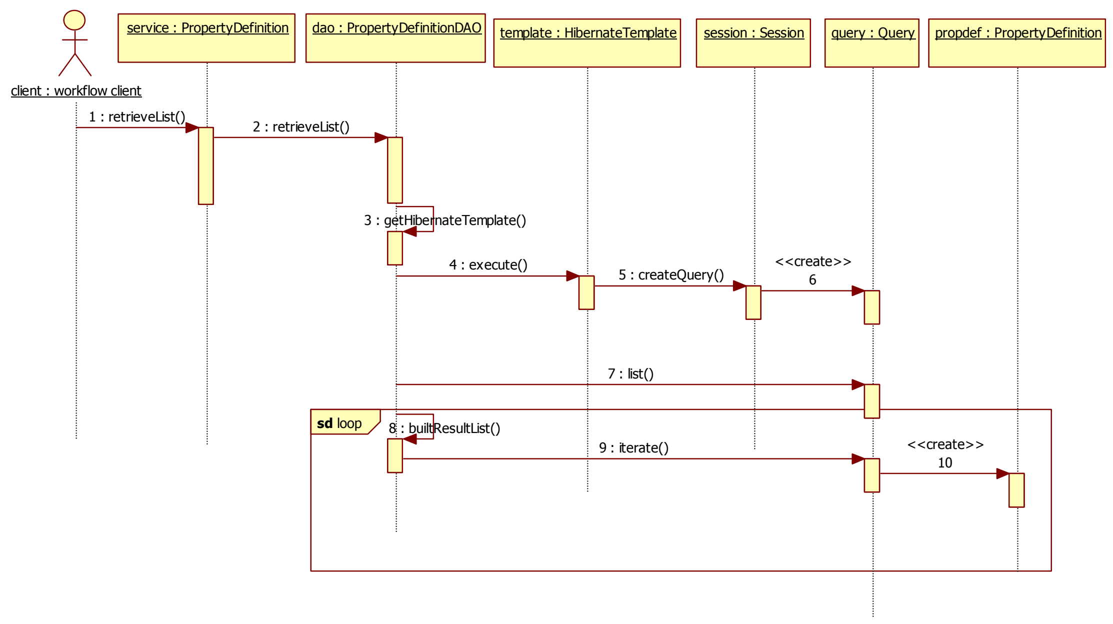


Figure 9 Retrieve Property List Sequence Diagram

Update Property Definition

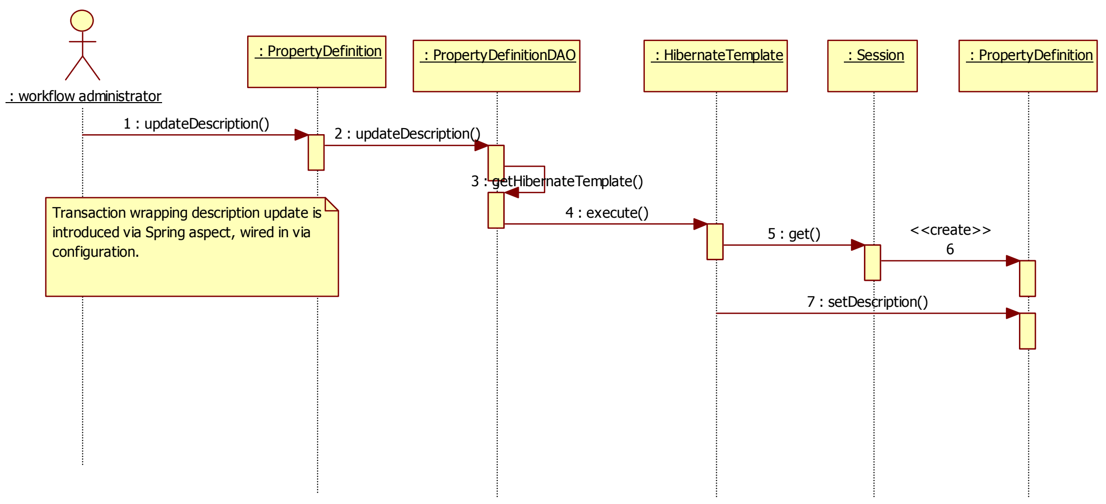


Figure 10 Update Property Definition Sequence Diagram

Execute Activity

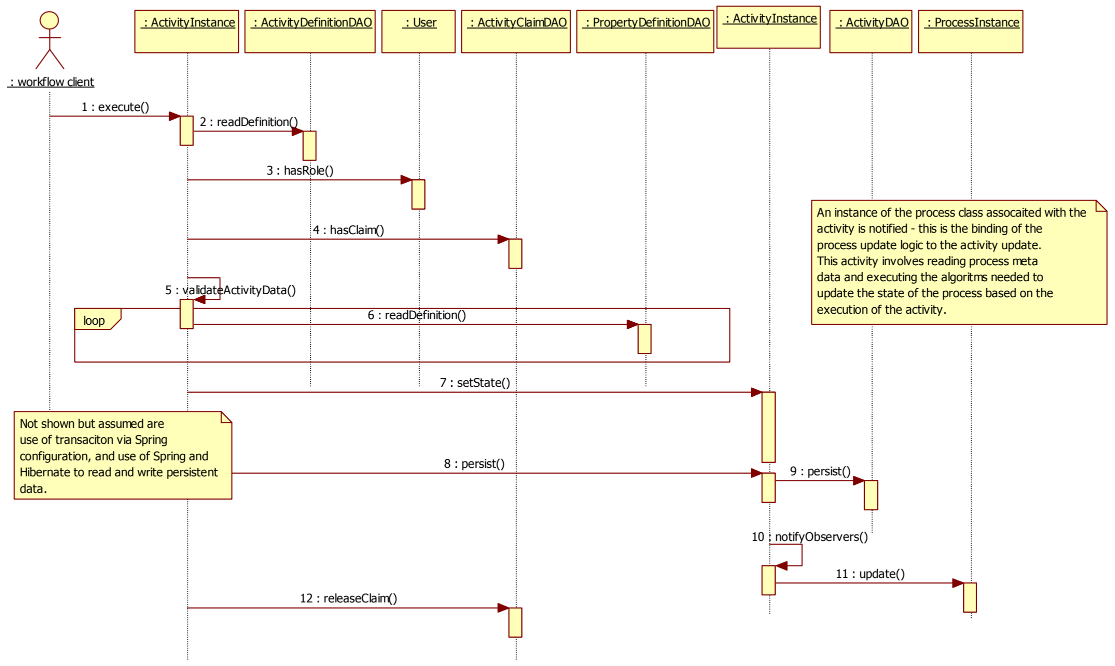

: workflow client
: ActivityInstance
: ActivityDefinitionDAO
: User
: ActivityClaimDAO
: PropertyDefinitionDAO
: ActivityInstance
: ActivityDAO
: ProcessInstance
1 : execute()
2 : readDefinition()
3 : hasRole()
4 : hasClaim()
5 : validateActivityData()
loop
6 : readDefinition()
7 : setState()
8 : persist()
9 : persist()
10 : notifyObservers()
11 : update()
12 : releaseClaim()
An instance of the process class assocaited with the activity is notified - this is the binding of the process update logic to the activity update. This activity involves reading process meta data and executing the algoritms needed to update the state of the process based on the execution of the activity.
Not shown but assumed are use of transaciton via Spring configuration, and use of Spring and Hibernate to read and write persistent data.

Figure 11 Execute Activity Sequence Diagram

Infrastructure and Deployment View

Server Issues

The ideal topology for scale and performance testing would mimic the target deployment environment – this would mean having test drivers hitting web services using a virtual service endpoint address, with a network load balancer spreading the requests over a number of physical servers.

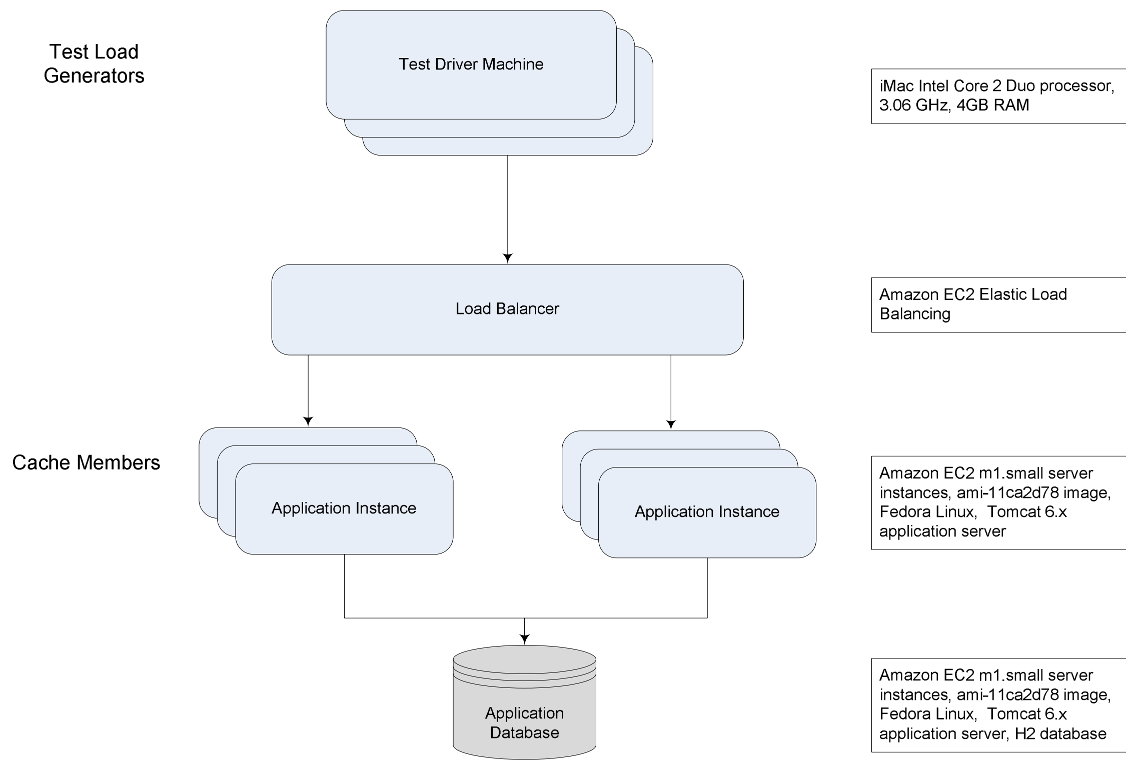

Test Load Generators
Test Driver Machine
iMac Intel Core 2 Duo processor, 3.06 GHz, 4GB RAM
Load Balancer
Amazon EC2 Elastic Load Balancing
Cache Members
Application Instance
Application Instance
Amazon EC2 m1.small server instances, ami-11ca2d78 image, Fedora Linux, Tomcat 6.x application server
Application Database
Amazon EC2 m1.small server instances, ami-11ca2d78 image, Fedora Linux, Tomcat 6.x application server, H2 database

Figure 12 Test Deployment Architecture

Amazon EC2 provides a low cost way to provision servers and load balancers based on commodity hardware and open source software. To avoid the license costs and complexity associated with enterprise software such as the WebSphere application server and the Oracle database, for this project open source software will be used for testing, specifically Apache Tomcat for the application server, and H2 for the database.

Network Issues

Amazon EC 2 does not allow IP multicast in its environment. Fortunately, most caching software (JBoss Cache included) supports the use of point to point communication as a configuration option.

Security View

Application Security

To reduce the scope of the project, we will not perform authentication, and trust the identity assertion of the web service caller, passed via a WS-Security user name token. Authentication can be layered on later as needed by configuring an interceptor to validate the identity claim on the user, via Spring Security in conjunction with an authentication provider [6].

For authorization, role-based access control will be used to control access to process instances and activities. The formal specification includes details on where role checking is performed.

Cache Cluster Security

Two issues to be considered in terms of cache cluster security are the need to ensure the confidentiality of the information being exchanged between cluster members, and the ability to ensure only authorized machines can join the cluster.

For this application, no sensitive information is being exchanged between cache members: only workflow metadata is being exchanged. Since the information does not need protection, especially given that all traffic will be on a network protected by a firewall, there is no need to use transport-level encryption to prevent eavesdroppers obtaining the information being exchanged by cluster members.

Cluster membership is more problematic. A malicious user could join the cluster and poison the cache with bad data, inject metadata to alter processes (for example, raise an automatic purchase approval amount from $75 to $10,000), launch a denial of service attack by flooding the cache with update requests to generate high replication volumes, and so on.

To secure group membership, and authorization protocol stack element (AUTH) can be wired into the protocol stack specification in the JGroups configuration file just below the group membership stack element (GMS), to allow only authorized group members into the group, with the identity of group members established via X.509 certs. For the purpose of the project, we will not use this protocol element to keep deployment and configuration as simple as possible, and will simply note this is an option for those cases where the additional security is desired.

Monitoring, Management, and Administration View

The primary mechanism for monitoring will be via logging. During construction phase, the significant log messages will be identified and documented such that an "ops doc" can be assembled, allowing someone in a support role to understand what error messages are significant, and what action needs to be taken based on an error message being logged.

Data View

The translation of the conceptual class or information model to a relational data model is straight-forward. In doing this, I mapped object attributes to columns, added identity columns that can be generated by the persistence layer or the database itself, and added tables for maintaining many to many relationships.

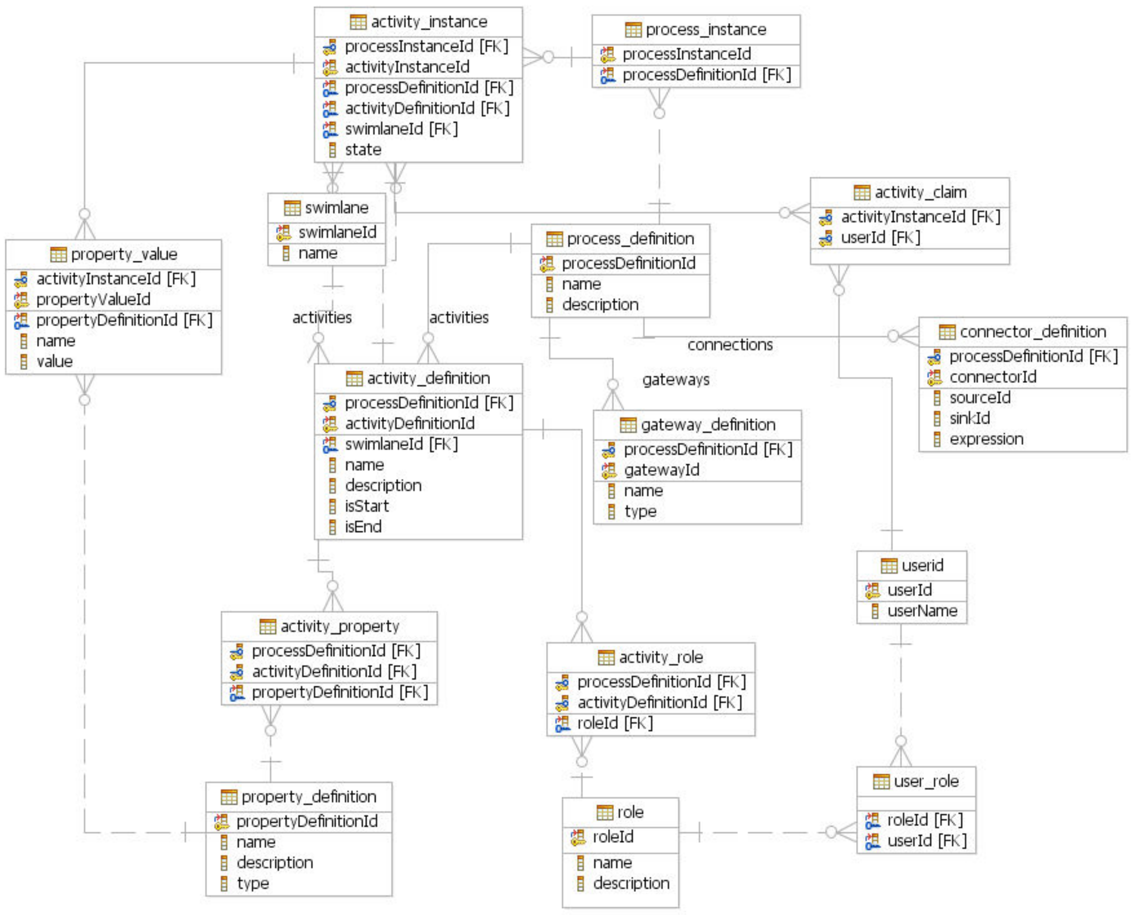


Figure 13 Database Structure

Justification of Non-Functional Requirements

This section presents the non-functional requirements, and how the architecture described in this document satisfies them.

QA1 - The solution may not degrade average response time or throughput for transactions involving cached data.	This will be accomplished via a combination of service implementation and cache configuration. Configuration options allow changing cache strategies (synchronous vs. asynchronous, replication vs. invalidation) and what's cached (entities and queries) to tune performance and scalability. Architectural risk is this will not be sufficient to address this quality attribute requirement.
QA2 - Functional correctness must be maintained when caching is enabled.	Second level cache controls where data is retrieved from (cache or database); program correctness determined by how that data is used.
QA3 - Application availability must not be compromised by the caching solution. The solution must provide the ability to survive failure of cluster members without	JGroup protocol implementation underlies cache replication and communication, and provides policies for dealing with crashed group members, group splitting and rejoining, and

compromising application availability or system correctness.	members joining and leaving the group.
QA4 - The solution shall provide a role-based security model to constrain access to the functions and data associated with the system based on roles.	To reduce scope this will not be fully implemented. However, this will be implemented for the execution services; extending this to all services is not an architectural risk.
QA5 – the solution shall provide basic authentication and identity management, where users are required to authenticate prior to accessing the system, and user identities associated with one or more roles.	This will not be fully implemented in the interest of appropriately sizing scope. Given the use of the Spring framework, introducing this via the Spring Security framework is a natural extension to the work in this project, and thus de-scoping this does not introduce architectural risk or risk invalidating any conclusions drawn from testing this implementation in terms of applicability to the target system.
QA6 – the solution must be scalable to 16 JVMs evenly distributed among 4 physically distinct servers, and reduces database requests for workflow metadata by at least 50%.	Use of JGroups with the appropriate caching strategy should make this goal achievable.
QA7 - data access must be consistent across all JVMs serving application requests. It is unacceptable for results produced by the application to differ based on the server or JVM servicing the application request. In other words the view of the data across all cluster members must be consistent.	Cache locking and replication strategy, coupled with the JGroups reliable multicast implementation, should make this an achievable goal.
QA8 - the solution must provide the ability to add and remove cluster members as needed without causing errors.	JGroups group membership protocol, coupled with appropriate cache size constraints and start up time configuration should make this possible.

Architecture Validation

Caching vs No Caching

The goal of this project was to produce system performance and scalability via the incorporation of a scalable cache solution, with the assumption that the system could be made to scale if the caching solution scaled.

To demonstrate the effect of caching vs read delays, I ran a transaction mix that instantiated a process 200 times, claiming and executing each of the 3 tasks in the process, plus retrieving task lists.

In the code, I added a Hibernate interceptor that intercepted property definition load. For loads that required a database read (cache misses in the case where caching was employed, all property read when caching was not used), I provided a hook to insert a configurable delay, and ran the above scenario for several delay values. The following chart shows the effect of the delay.

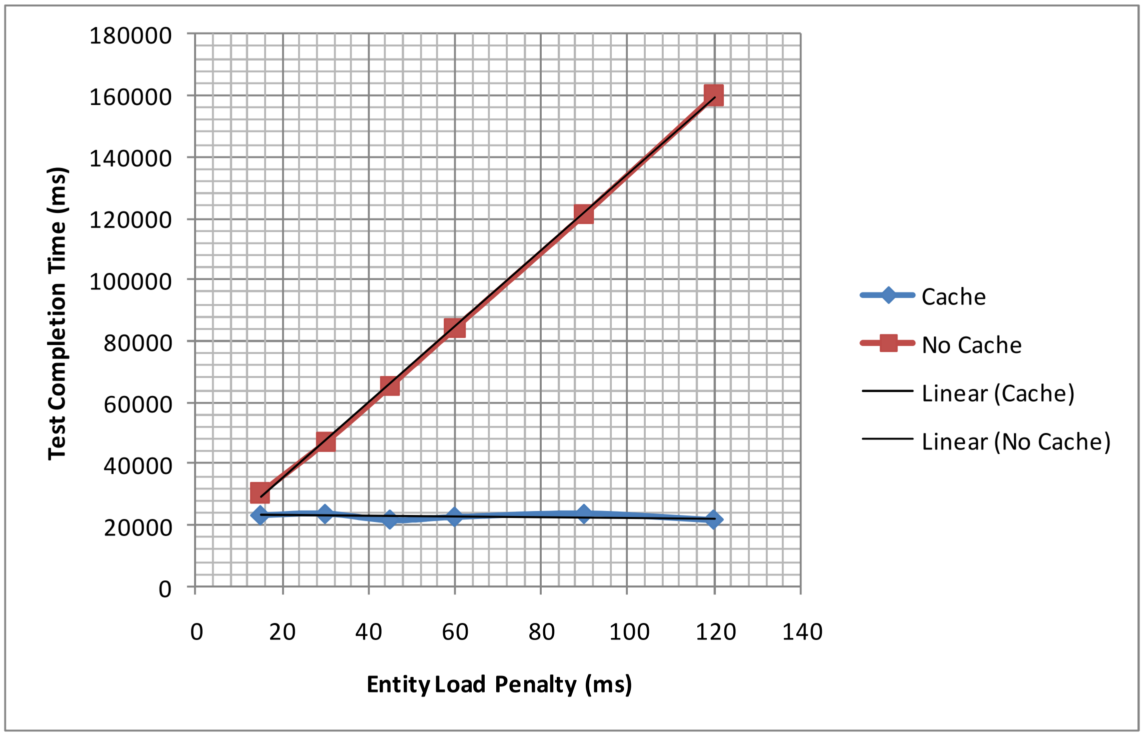


Figure 14 Effect of Read Penalty on Cache vs Non-Cache Use

From the above graph, it's clear that avoiding database read penalties via caching can produce more predictable response times.

Scalability

The architecture outline in part 1 was implemented and tested using the test topology shown in figure Figure 12. The following graph summarizes what was observed in scalability testing.

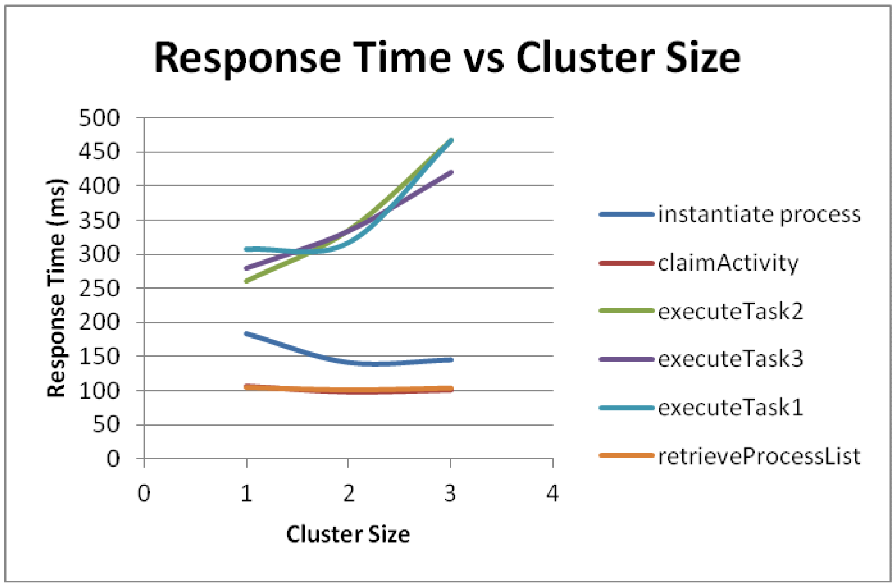


Figure 15 Response Time vs Cluster Size, Architecture 1

In this test scenario, one client process was used for each active cluster member, e.g. 1 client process (3 threads, 25 passes through the transaction script) for a 1 server cluster, 2 client processes (each with 3 threads doing the transaction script 25 times) for a 2 server cluster, and son on.

Noting the goal of this project is to scale to 16 nodes, we've run into trouble with only three nodes. From a caching perspective, the instantiate process transaction behaves nicely, as its reponse time stays roughly flat, based on the process definition data being cache and the reading of the data dominating the activity in the transaction.

However, overall system response time degrades significantly, dominated by the write heavy task execution transactions. The following table shows the summary data the graph is based on. Note the significant increase in response times as the cluster size is increased. If the system exhibited linear scalability the response times would remain flat. I would accept a 15% - 20% degradation as we reached a cluster size of 8 or more cluster members, but here we are seeing the execute task response times increasing roughly 70% by the time the third cluster member was added.

Throughput was also undesirable, departing significantly from the ideal of linear scalability when only the third cluster member was added:

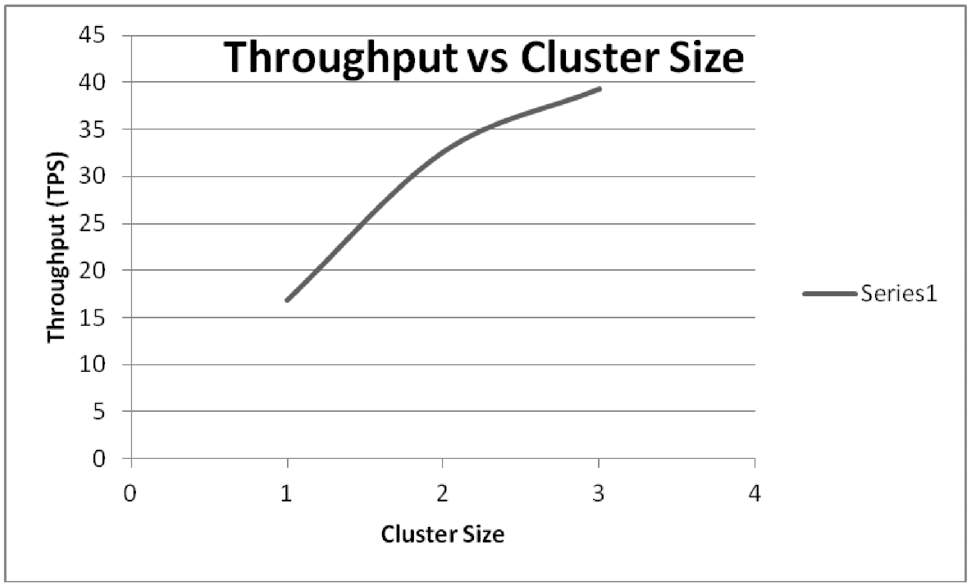


Figure 16 Throughput vs Cluster Size, Architecture 1

Based on the test results, it is clear the initial architecture has been invalidated.

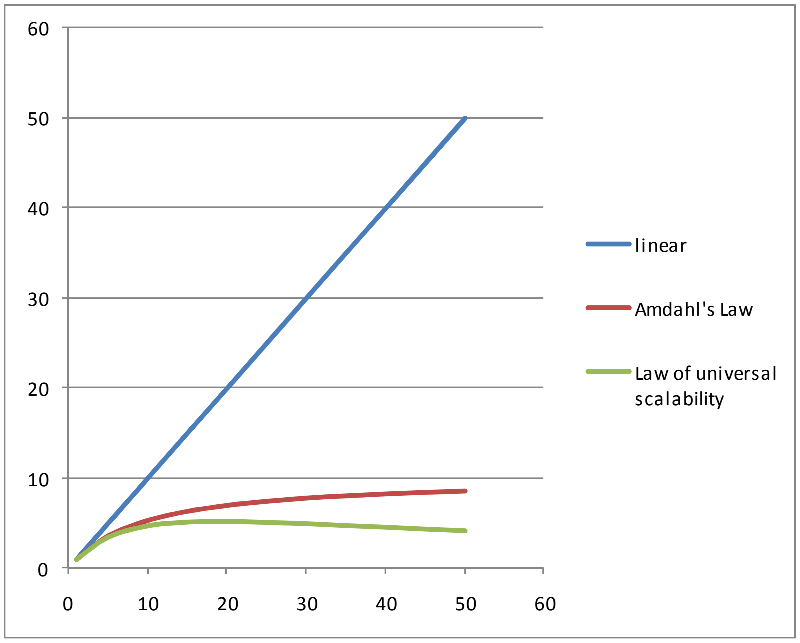
Part 2 – Revised Architecture

While caching can eliminate trips to the database and speed up reads (and make their response times more predictable), it was not a panacea for producing a scalable system. Digging into the matter of scalability, I learned of some models used to predict limits of system scalability in distributed systems.

The first was Amdahl's Law, which can be paraphrased as stating the speed up of and parallelized algorithm will ultimately be bounded by the serialized component of that algorithm [10]. This is relevant in the context of the system I am build in that while additional system nodes can be added to accommodate additional users or transactions, cache replication requires synchronization across all the nodes involved in a cluster, in effect serializing some of the processing between the logically independent nodes.

The second model was the Law of Universal Scalability, which in effect states Amdahl was an optimist as there is also a penalty to be paid to ensure data coherency, defined as all nodes in the system seeing the same view of the data at the same time [11].

The following plot shows the effects of Amdahl's Law and the Gunther's Law of Universal Scalability for an arbitrary workload:



Gunther points out a processor (or a node) will not be doing any useful work if it must:
"

- *wait for a bus transfer from another processor or memory*
- *wait for I/O completion*
- *serialize on an RDBMS lock or latch*

"

While a caching solution may reduce or eliminate much of this on a single node, from an overall system perspective the centralized database is the bottleneck if all transactions must interact with the database, with network communication and database synchronization/serialization in effect serializing much of the overall system workload.

Scale testing the architecture detailed in part 1 showed that write heavy transactions were problematic in terms of overall system performance. While it could be argued caching improved transactions dominated by database reads, it did not provide a solution from an overall system scale and performance perspective.

To address this, instead of basing the architecture on the traditional 3-tier architecture with cache enhancements attempting to reduce the serialized load on the database, a data grid architecture will be used.

The basic idea of the data grid is to concentrate reads and write in the data grid, with data distributed across grid members, where a piece of data belongs to one member of the grid (replicated to one or two other grid members to avoid data loss in the event of a grid member crashing). Data in the grid is then written to the store periodically, a scheme known as write-behind caching.

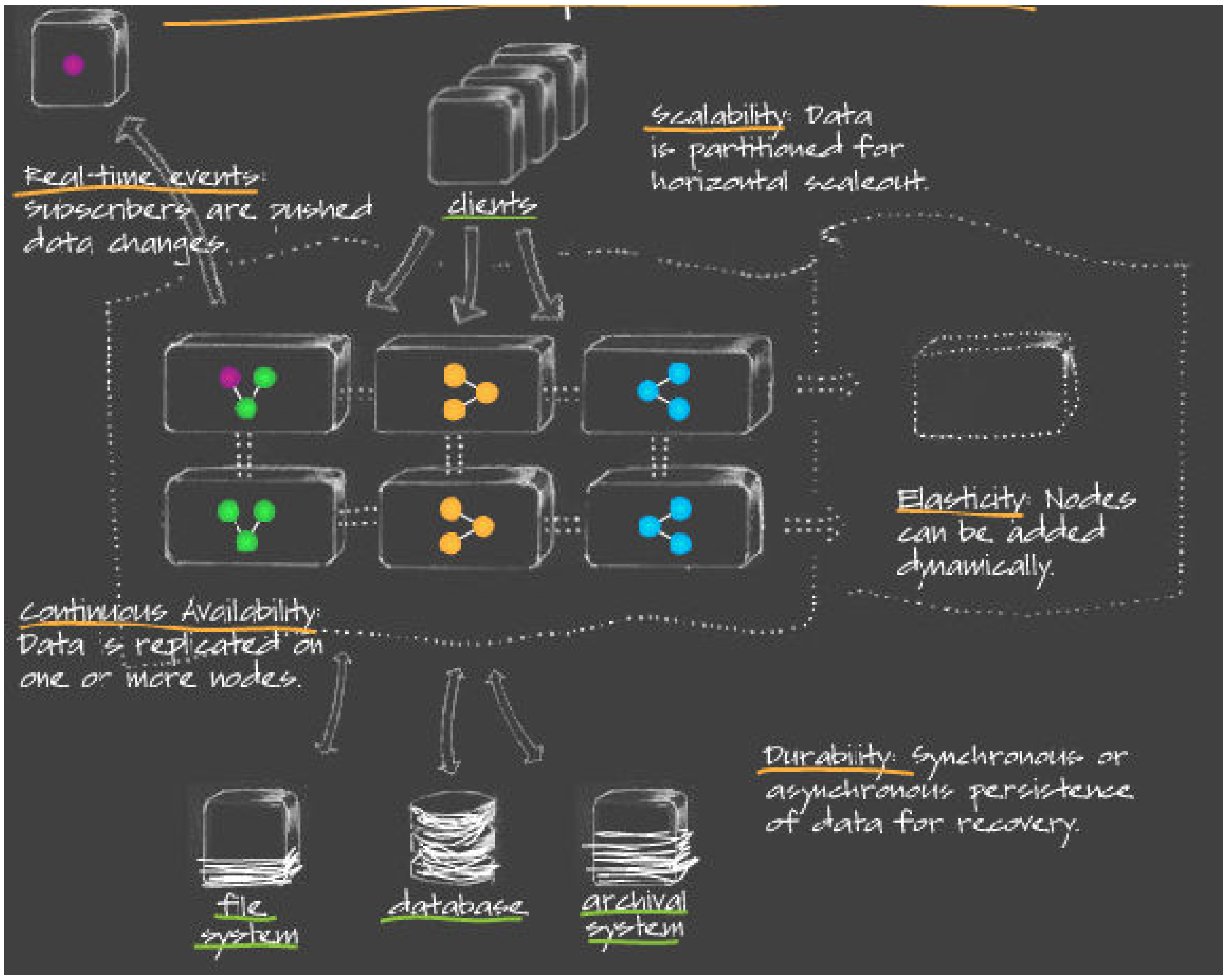

Real-time events: subscribers are pushed data changes.
clients
Scalability: Data is partitioned for horizontal scaleout.
Elasticity: Nodes can be added dynamically.
Continuous Availability: Data is replicated on one or more nodes.
Durability: Synchronous or asynchronous persistence of data for recovery.
file system
database
archival system

Figure 17 Data Grid Architecture (from http://www.gemstone.com/products/gemfire)

From an overall system perspective, this provides some important benefits:

- Using a grid with a limited number of nodes owning data puts a ceiling on the number of nodes that need to be coordinated when replicating data. Through configuration we can simply make a single backup, so at most two nodes need to coordinate updates to cached data.
- Using a grid backed by a relational database, with persistence to the database done using an asynchronous write behind strategy, the serialization and coherence penalties can be paid outside the work associated with an interactive transaction.

For the updated architecture, I used Hazelcast as my IMDG [8]. Hazelcast provides a Java collections programming abstraction, with the IMDG software managing the data replication and data eviction on behalf of the user as specified in configuration.

Services were rewritten to interact solely with the grid, with database access configured using Hazelcasts Loader and Store interfaces, plugged in via configuration. [9] Reading metadata would look for the data in a map; if the data was not in the grid Hazelcast would use the Hibernate DAOs written in the first architecture's implementation to load the data into the grid, at which point all subsequent reads went against the grid.

Application reads for volatile data also went against the grid, as opposed to reading directly against the database.

Application writes are done using a write behind strategy: data is written to the grid, then persisted asynchronously in batches via Hazelcast's write behind.

Based on the above, the application software layers looks like this:

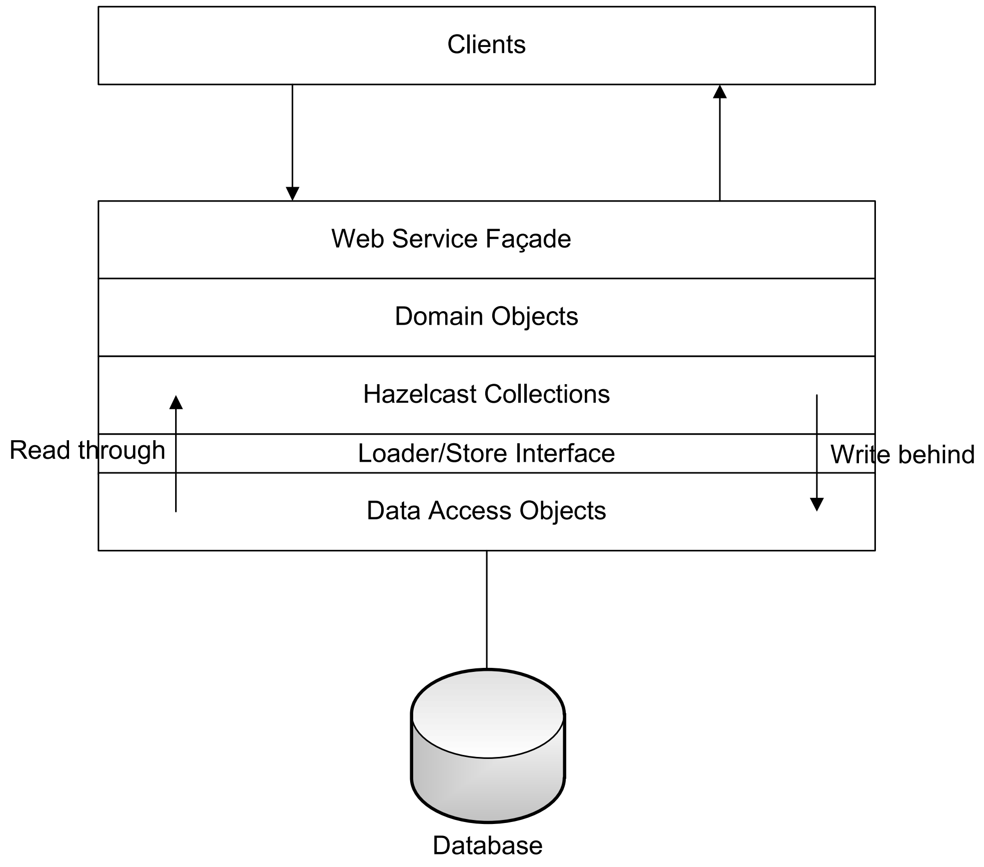


Figure 18 Updated Software Layers

Based on the layering and the brokering of database access by the IMDG, the package dependencies now look like this:

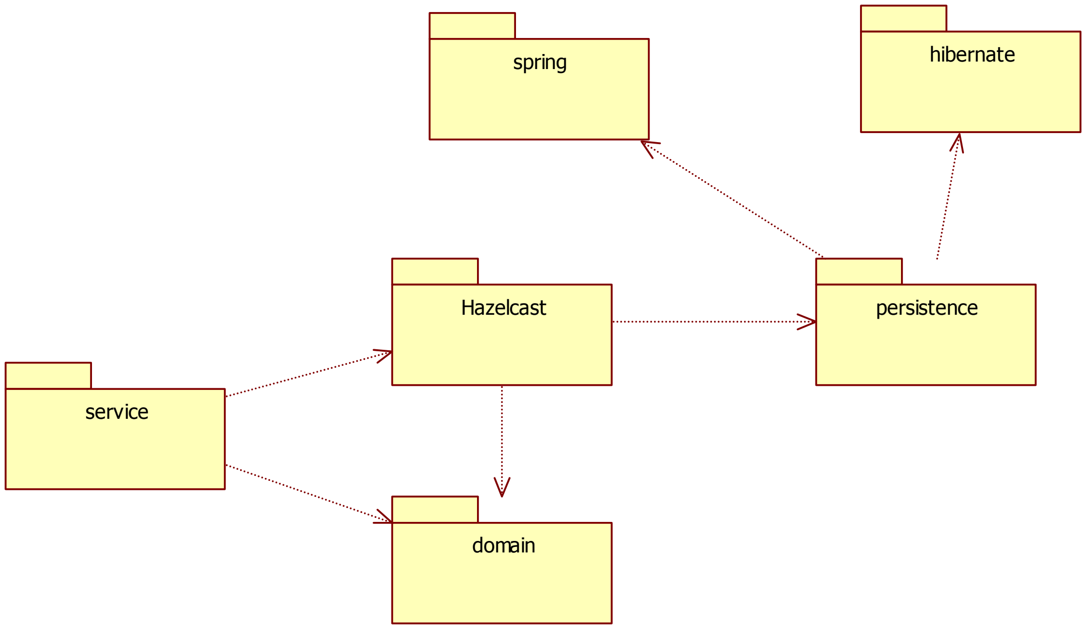
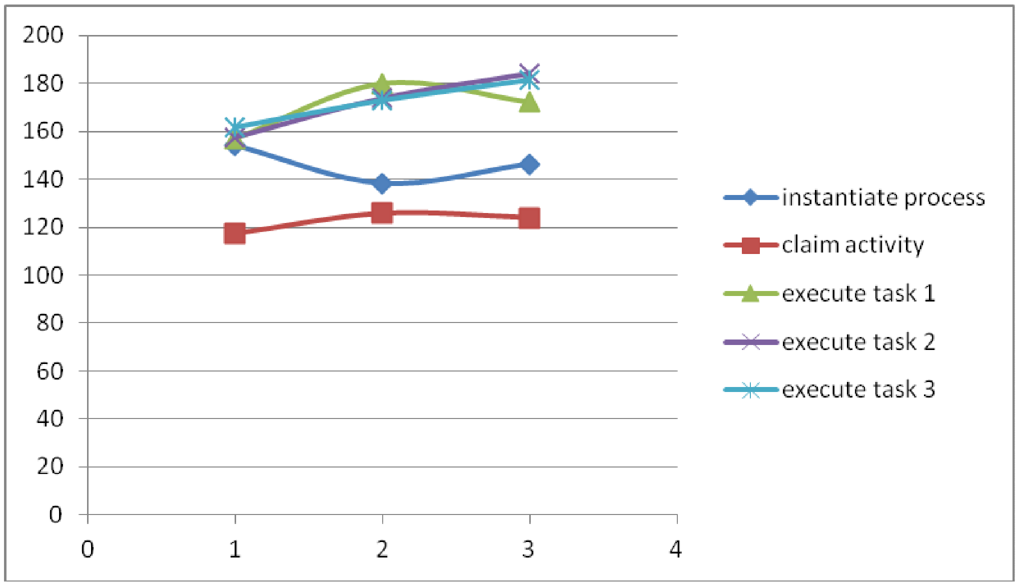


Figure 19 Updated Package Dependencies

Architecture Validation

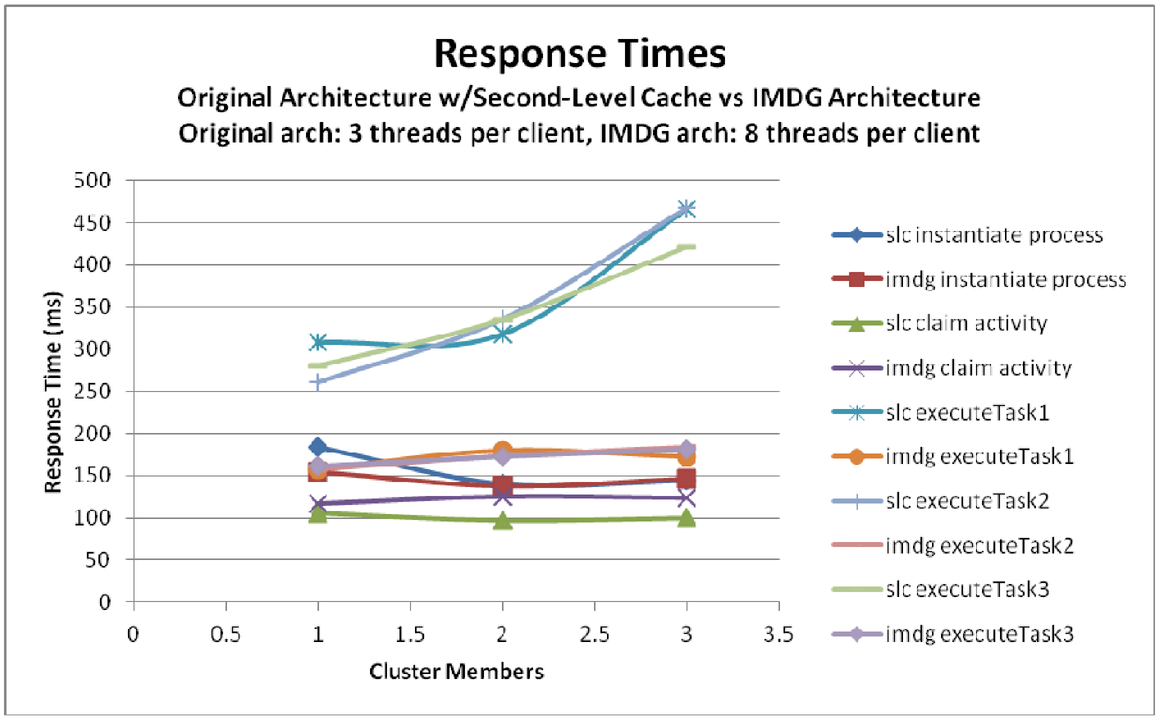
The architecture revision has shown better performance and scalability characteristics. From a testing performance, cluster size was evaluated running more client threads per test process (8 instead of 3) and more test scenario iterations (100 instead of 25), with response times under greater load being significantly faster.

The following plot summarizes the scalability testing:



Here we see the response times grouped closer together, with more similarity between read trends and write trends. While response times are growing, it is likely that due to cluster data partitioning and a cap on the number of nodes data is replicated to, it will approach a reasonable leveling off point. Furthermore, near caching in the grid can be used to reduce the amount of traffic read between cluster members.

The following plot combines the old and new architecture plot. Again, *note this is not a true comparison as the new architecture is running nearly 3 times as many client threads per test process and executing 4 times the number of transactions per thread.*



Note the part of the chart exhibiting unacceptable response time increase belongs to the orginal architecture.

References

[1] Vision Document: MSE Project, Doug Smith, http://people.cis.ksu.edu/~dougs/Site/Phase1_Artifacts_files/vision.pdf

[2] Software Architecture Document Guidelines, v 0.1, Simon Brown, http://www.codingthearchitecture.com/files/software-architecture-document-guidelines-v0.1.pdf

[3] Formal Specification Document: MSE Project, Doug Smith

[4] Reliable Multicasting with the JGroups Toolkit, Bela Ban, http://www.jgroups.org/manual/pdf/manual.pdf

[5] Beocat: K-State's Beowolf Cluster, http://www.k-state.edu/beocat/

[6] Spring Security Reference Documentation, http://static.springframework.org/spring-security/site/reference/pdf/springsecurity.pdf

[7] The Rational Unified Process: An Introduction, Philippe Kruchten, Addison Wesley Professional, 2003

[8] Hazelcast IMDG – www.hazelcast.com

[9] Hazelcast Loader and Store integration – see http://www.hazelcast.com/documentation.jsp#MapPersistence

[10] "Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities", Gene Amdahl, AFIPS Conference Proceedings (30): 483–485.

[11] "A Simple Capacity Model of Massively Parallel Transaction Systems", N. J. Gunther, CMG National Conference, 1993