SAND2014-20571R

US NDC Modernization

SAND-xxxx
Unclassified Unlimited Release
December 2014

US NDC Modernization Iteration E2 Prototyping Report: OSD & PC Software Infrastructure

Version 1.1

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

SAND-xxxx
December 2014

US NDC Modernization Iteration E2 Prototyping Report: OSD & PC Software Infrastructure

Ryan Prescott
Ailsa Chiu
Bernard L. Marger

Version 1.11
Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185

ABSTRACT

During the second iteration of the US NDC Modernization Elaboration phase (E2), the SNL US NDC Modernization project team completed follow-on COTS surveys & exploratory prototyping related to the Object Storage & Distribution (OSD) mechanism, and the processing control software infrastructure. This report summarizes the E2 prototyping work.

REVISIONS

Version	Date	Author/Team	Revision Description	Authorized by
1.0	9/19/2014	US NDC Modernization Team	Initial Release	M. Harris
1.1	12/19/2014	IDC Reengineering Team	IDC Release	M. Harris

TABLE OF CONTENTS

1. OVERVIEW

The US NDC Modernization project statement of work identifies the definition of a modernized US NDC system architecture as a key project deliverable. As part of the architecture definition activity, the Sandia National Laboratories (SNL) project team has established an ongoing, software prototyping effort to support architecture trades and analyses, as well as selection of core software technologies.

During the second iteration of the Elaboration phase (E2), spanning Q3 – Q4 FY2014, the prototyping team developed follow-on COTS surveys and exploratory prototypes related to the Object Storage & Distribution (OSD) mechanism and processing control software infrastructure. This report summarizes E2 work and discusses the path forward.

2. FOCUS AREAS

Prototyping work to date has focused on three areas of the system, including the OSD mechanism, processing control software infrastructure, and user interface framework (UIF). This report addresses only the OSD and processing control-related activities. E2 user interface prototyping work is documented in a separate report [1].

2.1. OSD

The OSD mechanism[1] provides application programming interfaces (APIs) for access to data stored in the system database as well as distribution of data among processing components. Figure 6 in Appendix A illustrates the components of the OSD. The following goals and constraints have been identified for this mechanism:

- Minimize dependencies between the system application/research tools and underlying data storage solutions.
- Decouple the application data model from the physical data model (e.g. DB schema).
- Provide a query language that is independent of the underlying data storage solution.

[1] As defined in the US NDC System Architecture Document (SAD), a mechanism represents a basic service required by many subsystems across the system. Examples include the Processing Sequence Controller, and Object Storage & Distribution.

- Provide optimizations as needed to support system performance requirements – e.g. in-memory caching.
- Prefer Open Source Software (OSS) and other Commercial Off-The-Shelf (COTS) solutions to custom software development where available.
- Prefer solutions based on open standards.

2.2. Processing Control

The processing control software infrastructure[2] provides for the definition, configuration, execution and control of system processing components, supporting both automated and interactive analysis processing. The following design goals and constraints have been identified for the processing control software.

- Provide a fault-tolerant, scalable processing model
- Provide or support a means for defining and configuring processing sequences
- Provide an interface abstraction to facilitate integration of new processing algorithm implementations
- Provide a messaging framework for communication of data and processing control information among processing components
- Support processing components implemented in the languages to be defined for the modernized system
- Prefer Open Source Software (OSS) and other Commercial Off-The-Shelf (COTS) solutions to custom software development where available.
- Prefer solutions based on open standards.

3. OSD PROTOTYPING

3.1. E1 Background

The E2 OSD prototyping effort represents a progression of the activities completed in E1. A brief summary of the E1 OSD work is provided below for context.

E1 OSD prototyping focused on evaluation of COTS object-relational mapping (ORM) software for use in developing data access APIs for the system's data storage solution. Exploratory prototypes were developed for two of the ORM solutions surveyed – Hibernate (Java) and ODB (C++). Ultimately Hibernate was

[2] The term software infrastructure is a more general term for a collection of supporting services that may include one or more mechanisms.

selected for further evaluation in E2 based on a project decision designating Java as the primary application development language for the modernized system.

3.2. E2 Prototyping

The following activities were completed in E2:

1. Follow-on prototyping of the Hibernate ORM supporting OSD data access
2. Evaluation of middleware for Python access to Java ORM APIs supporting OSD data access from external scripting environments
3. Survey of messaging technologies supporting OSD data distribution, as well as processing control software infrastructure
4. Survey of data grid solutions for OSD data distribution

The following sections describe each of these activities in further detail.

3.2.1. Hibernate ORM

As indicated in Appendix A, an ORM solution will be used to implement Data Access Objects (DAOs), providing access to stored data both within the system, and from external (e.g. research) environments. In E2, the team developed an expanded Hibernate-based ORM prototype demonstrating storage, modification and query-based retrieval of event hypothesis, signal detection and waveform entity objects from an underlying Oracle DBMS instance. This work was completed as part of a summer student intern project.

The E2 work demonstrated Hibernate's automated conversion between application entity objects and table data stored in the underlying database. The prototype demonstrated both of Hibernate's object-to-relational mapping approaches: XML specifications, and in-code annotations. Both were found to be straightforward to implement, and significantly reduced the effort required to develop persistence APIs for the selected entities (event hypothesis, signal detection & waveform). It should be noted that the in-code annotation mapping approach is considered to be the best practice, given that it results in a simpler, less verbose interface relative to XML-based mappings.

A remaining concern for Hibernate (and COTS ORM solutions in general) relates to access performance. It is not yet clear whether Hibernate can meet the system's read/write performance requirements for worst-case scenarios under load. At issue is the fact that Hibernate automatically generates SQL statements for CRUD operations, which may not be optimized for performance given the system's database schema. For these cases, Hibernate allows the generated SQL to be overridden with user-specified statements. In E3, the team will assess Hibernate performance against stressing queries/data sets. In the event that

Hibernate cannot meet performance requirements, alternate approaches (e.g. custom ORM built on JDBC) will be pursued.

3.2.2. Python Data Access

Access to stored system data via the OSD from external scripting environments will be required in the modernized system. Python has been identified as a candidate scripting language. In order to investigate Python-based OSD access, a summer intern project evaluated Python/Java interoperability middleware options. Two types of solution were considered as part of the project:

1. Jython, formerly JPython is an alternate implementation of the Python language specification written in Java (the Python reference implementation is written in C). Jython programs are compiled (statically or dynamically) to Java bytecode, which runs on the Java Virtual Machine (JVM). Jython is therefore able to provide direct access to Java classes and APIs. The primary disadvantage of Jython is that it is an alternative Python implementation. As such, it less widely used, and not as well supported as the reference implementation.

2. JPype and Py4J are middleware solutions providing access to Java classes from within Python programs using Java Native Interfaces (JNI). Both solutions provide Java type mappings (including primitives, strings, collections, user-defined types & classes, etc.) as well as support for Java language features such as exceptions, threading & synchronization. Py4J is under active development, with the latest version (0.8.2) released in July of this year. JPype's current development status is less clear; the most recent version was released in 2011.

   The primary advantage of these solutions is that they provide robust Java interoperability with the reference Python implementation. The primary disadvantage is the potential for additional latency introduced by the addition of JNI interfaces. It is unclear whether latency is truly an issue however. The providers of JPype for example, argue that Java software accessed via JPype is likely to be faster than an equivalent Python program given JVM optimizations for code execution speed relative to Python. [2]

Prototypes were developed using both approaches demonstrating basic access to Java entity classes, including event hypothesis, signal detection & waveform classes. Although both approaches were found to be functionally similar, the middleware approach (JPype, Py4J) was determined to be the better solution because it supports the reference Python implementation. Further prototyping of this solution will be completed as needed in future iterations.

3.2.3. Messaging Software

Inter-process communication (IPC) is a key underlying capability required in distributed computing architectures for the exchange of data, control, and status information between independent processing residing on one or more hosts.

Although there are numerous low-level IPC technologies available – files, memory-mapped files, shared memory, signals, semaphores, pipes, named pipes, etc. – the survey focused on message-based solutions for the following reasons.

- They provide a generalized communication infrastructure supporting information exchange between processes distributed over a network.
- They provide higher-level abstractions, insulating applications from details of the underlying communication stack.
- They provide a flexible set of communication patterns supporting a wide array of application architectures.
- They have become the dominant technology for modern distributed systems.

Appendix C provides a brief overview of relevant concepts, standards and use cases related to messaging software.

3.2.3.1. OSD & Processing Control Messaging

Messaging software COTS can be applied to support both the OSD data distribution requirements, as well as those of the processing control software infrastructure.

3.2.3.1.1. OSD Data Distribution

As currently defined in the US NDC architecture, the OSD mechanism provides publish/subscribe-based distribution of data across processes, including both persistent data stored in the database, and transient data maintained in memory. Messaging frameworks provide a robust, widely-adopted solution space that can be used to support this capability. In order to address the full set of use cases, multiple distribution design patterns will likely be needed. Examples include direct transmission of application data between processes (see Figure 1), and indirect distribution of data via the database or distributed caching middleware with message-based notification of subscribers (see Figure 2).

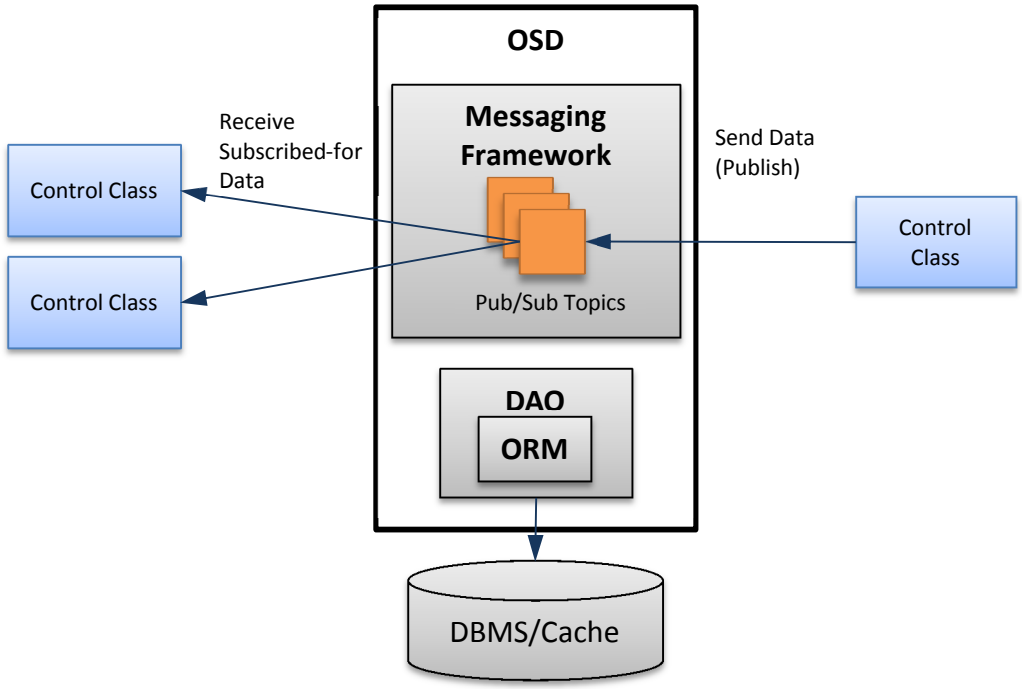

Figure 1. Example Messaging for Data Distribution

Application components publish data directly to other components who subscribe via publish/subscribe topics. The OSD provides data serialization and messaging support.

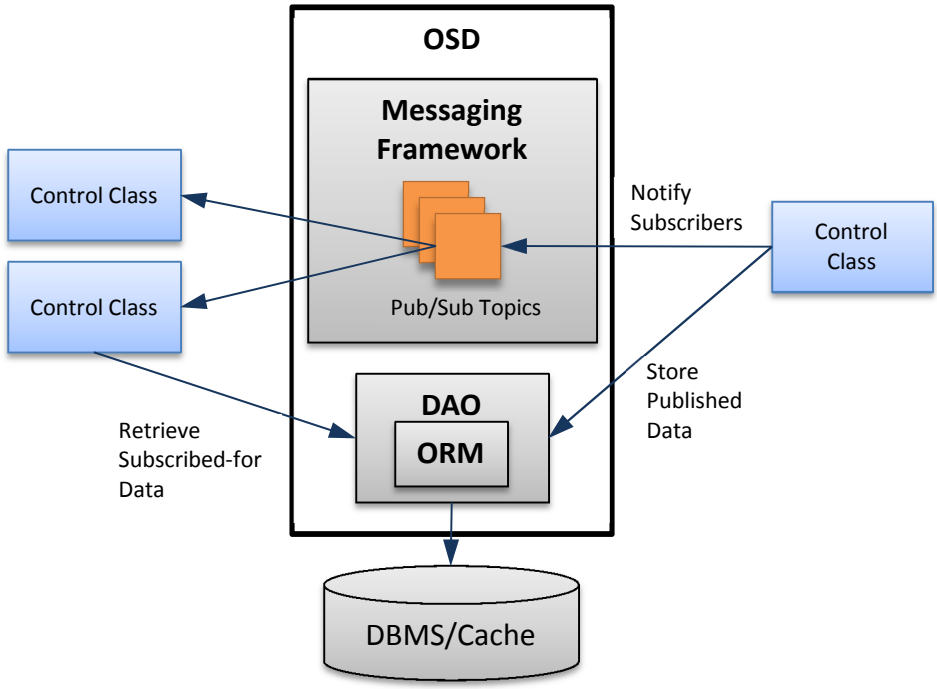

Figure 2. Example Notification Messaging for Data Distribution

Application components publish data indirectly to other components by storing data in the database or middleware cache via Data Access Objects (DAOs) and publishing notification messages to subscribers. Other components subscribe for data through messaging publish/subscribe topics. Upon receipt of notification messages, subscribers retrieve the indicated data from the database/cache.

3.2.3.1.2. Processing Control Messaging

Messaging frameworks can be used as part of a processing control solution to communicate tasking and other control information between processing components. The example in Figure 3 shows messaging between the *Process Sequence Control* mechanism and control classes in order to distribute processing sequence tasks and collect processing status.

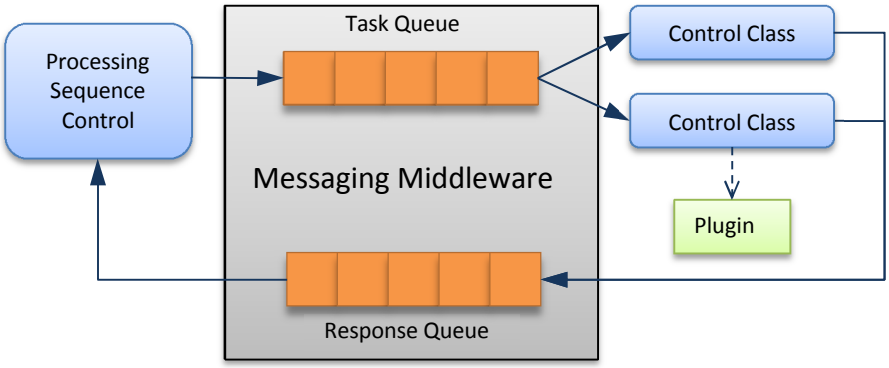

Figure 3. Example Messaging for Task Distribution

3.2.3.2. Messaging Framework Survey

Several popular, open-source frameworks were surveyed as part of the E2 work, including Apache ActiveMQ, Apache Apollo, Apache QPid, RabbitMQ, ZeroMQ, and RTI DDS. The following sections provide an overview of each framework, as well as high-level comparisons & conclusions. Table 1in Appendix B summarizes the survey findings.

3.2.3.2.1. ActiveMQ

ActiveMQ is a highly flexible, open-source, brokered messaging solution providing broad support across a number of languages, standards and transport protocols. It is a top-level project of the Apache Software foundation, and has been under active development since 2003. It is released under the Apache License 2.0. Commercial support is available from a number of third-party vendors. [3]

Although ActiveMQ uses its own OpenWire[3] protocol by default, it also supports AMQP, STOMP, XMPP, REST & WS Notification, among others. ActiveMQ supports various transport protocols, including in-memory, TCP, SSL, NIO, UDP, multicast, JGroups and JXTA. Clients are available in a variety of languages, including Java, C, C++, C#, Python, Ruby, Perl & PHP. [4] It is fully compliant with the JMS 1.1 standard, and integrates with Spring as the JMS provider via the JMSTemplate. ActiveMQ does not provide cross-language data serialization; this is the responsibility of the client application.

ActiveMQ provides at-least-once processing guarantees using a combination of message acknowledgment, message queue persistence, and broker clustering (see Figure 8 in Appendix C).

[3] OpenWire is a cross-language, binary wire protocol developed as part of the ActiveMQ project.

ActiveMQ is used for messaging in several Enterprise Service Bus (ESB) implementations and other Service Oriented Architecture (SOA) frameworks, including MuleESB, Apache ServiceMix, Apache CXF, & Apache Camel.

ActiveMQ provides a number of additional features, including the following.

- Advanced features to support scalability and load balancing.[4]
- Routing features such as content-based message filtering & message prioritization
- Transaction support
- Security features, including authorization using a custom access controls mechanism, authentication using the Java Authentication and Authorization (JAAS) framework, and encryption & certificate management using SSL
- JDBC-based RDBMS integration for message persistence
- Broker management and administration support using the Java Management Extensions (JMX)

3.2.3.2.2. Apollo

Apollo is a subproject of Apache ActiveMQ that seeks to improve upon ActiveMQ broker performance & scalability, employing an alternate threading and message dispatching architecture to eliminate synchronization inherent in parts of the ActiveMQ broker. The intent is to better leverage high core counts on modern processors. [5] Apollo was first released in early 2012. It currently supports only a subset of the features available in ActiveMQ; STOMP is the only protocol currently available. Apollo is released under the Apache License 2.0.

The Apollo subproject describes itself as the next generation of ActiveMQ messaging, providing throughput and scalability improvements targeting future enterprise messaging needs. The Apollo project released benchmark results in 2011 showing favorable performance relative to ActiveMQ, RabbitMQ, & HornetQ for the scenarios addressed [6].

[4] ActiveMQ connection pooling, *Message Groups*, *Virtual Topics*, *Wildcard Topics* and *Composite Destinations* provide a great deal of flexibility in designing high-performance messaging systems with built-in load balancing & failover. [4] Optional message persistence is provided through a replicated LeveDB store.

3.2.3.2.3. QPid

Apache QPid is an open-source messaging solution implementing the AMQP standard. It is a top-level apache project under active development since 2005. It is released under the Apache License 2.0.

The Qpid project includes broker implementations in Java & C++, with client APIs available in a variety of languages including C, C++, Java, Python, Perl, Ruby, PHP & C#. QPid also provides a JMS 1.1 compliant client, as well as a Java Connecter Architecture (JCA) resource adapter for Java EE applications. QPid does not provide cross-language data serialization; this is the responsibility of the client application.

QPid provides at-least-once processing guarantees using a combination of message acknowledgment, message queue persistence, and broker clustering (see Figure 8 in Appendix C). QPid provides a number of additional features, including the following.

- Message transaction support
- Routing features such as content-based message filtering & message prioritization
- Security features, including authorization via ACLs, authentication using the Simple Authentication & Security Layer (SASL) framework, and encryption & certificate management using SSL
- RDBMS integration for message persistence in the broker
- Broker management and administration support using the Java Management Extensions (JMX) for the Java broker and the QPid Management Framework (QMF) for the C++ broker

3.2.3.2.4. RabbitMQ

RabbitMQ is a very popular[5] open-source message oriented middleware solution that implements the AMQP standard. It has been under active development since

[5] Interest in RabbitMQ as measured using Google Trends search term frequency is the highest among the messaging solutions surveyed, having recently surpassed ActiveMQ in 2013. See Figure 7 in 6.Appendix D for the Google Trends data.

2008, and is currently managed by Pivotal Software[6], which provides commercial support. RabbitMQ is released under the Mozilla Public License.

RabbitMQ includes clients for Java, C# and Erlang. Many other clients, including C/C++, Perl, Python, PHP are available from third-party projects. The Spring framework provides an integration with RabbitMQ via Spring AMQP. A command-line interface is also available. RabbitMQ does not provide cross-language data serialization; this is the responsibility of the client application.

RabbitMQ provides at-least-once processing guarantees using a combination of message acknowledgment, message queue persistence, and broker clustering (see Figure 8 in Appendix C). A number of additional features are available, including the following:

- Transaction support
- Security features, including authorization using a custom access controls mechanism, authentication using the Simple Authentication & Security Layer (SASL) framework, and encryption & certificate management using SSL
- Custom message persistence
- Routing features such as content-based message filtering & message prioritization

3.2.3.2.5. ZeroMQ (also øMQ or zmq)

ZeroMQ is a high-performance messaging library designed to support scalable applications (either concurrent or distributed). It is free, open-source software released under the GNU Lesser Public License, and developed by iMatix, which provides commercial support. Stated design goals for ZeroMQ include simplicity, performance and scalability. Unlike most other solutions surveyed, it employs a broker-less architecture.[7] [7]

ZeroMQ provides a several messaging patterns including exclusive pair (single pair point-to-point), publish/subscribe, request/response and push/pull (parallel task distribution & collection). It supports a number of network transports (TCP, TIPC, SCTP, PGM, etc.), as well as an in-process transport supporting multi-threading via the ZeroMQ messaging model.

[6] Pivotal Software, Inc. is a software and services company founded in 2013 and currently based in San Fancisco, CA.

[7] A broker-less architecture optimizes for performance at the expense of anonymity between sender and receiver - the network address of either the sender or receiver must be statically known by the other.

Messaging clients are provided in C, Python, Java, .NET, Ruby, PHP, Perl, and Erlang. ZeroMQ does not provide cross-language data serialization; this is the responsibility of the client application. Authentication and encryption are provided via the CurveZMQ custom protocol.

3.2.3.2.6. RTI DDS

RTI DDS is a robust, scalable real-time messaging framework implementing the DDS standard. It is developed by Real-Time Innovations (RTI), which provides both open-source and commercial versions. The open-source version is released under a limited open-source license developed by RTI called the Infrastructure Community License (ICL). The ICL allows distribution of the core RTI DDS product source code within an Infrastructure Community (IC), which RTI defines to be an organization with the goal of establishing common software infrastructure across projects, systems, etc. [8]

RTI DDS is designed for low-latency, high-throughput and deterministic quality-of-service under load. Current marketing for RTI DDS is targeted to the Internet of Things (IoT) and other real-time architecture domains. As with ZeroMQ, RTI DDS uses a broker-less peer-to-peer connection model in order to minimize latency. It supports multiple underlying transports, including for example, TCP, UDP, multicast & shared memory.

As defined in the DDS standard, RTI DDS provides fine-grained control over messaging quality-of-service, addressing both latency and message delivery guarantees. Optional message persistence supports asynchronous transmission to clients even if the sender is no longer available.

DDS and JMS messaging APIs are provided as part of the core library. Integration services are available in the commercial edition for web integration (via REST & SOAP), and relational database management systems. Clients are provided supporting C, C++, Java & C#. RTI provides for serialization of message data using code generation based on the DDS interface description language (IDL). RTI DDS provides a number of additional features, including the following:

- Security features, including access controls, authentication, and encryption via OpenSSL and TLS
- Facilities for recording and replaying message traffic (commercial edition)
- Sophisticated monitoring, modeling and administration tools to support configuration, tuning & troubleshooting (commercial edition)

RTI DDS markets itself as the most prevalent of the DDS-based solutions. [9] It has a broad customer base across a number of industries including, for example, healthcare, defense, energy, transportation, industrial and communication. [10]

3.2.3.3. Summary Comparison

All of the solutions surveyed provide flexible support for various messaging patterns, as well as scalability, reliability and security features.

All of the solutions surveyed provide support for the most common development languages, including C, C++, Java & C#. Except for DDS, all solutions also support Python, Perl & Ruby.

Two messaging standards are dominant among the solutions surveyed:

1. **AMQP** - ActiveMQ, Apollo, QPid, and RabbitMQ all implement the Advanced Message Queuing Protocol (AMQP) standard

2. **DDS** - RTI DDS implements the Data Distribution Service (DDS) standard

The DDS standard is specialized for low-latency, high-throughput, real-time systems where fine-grained control over quality of service policies is required. Many of these features are likely not critical drivers for the system, and come at the expense of greater complexity compared with the AMQP solutions. AMQP is a more general-purpose standard emphasizing flexible messaging patterns, performance, interoperability, reliability and security.

With the exception of RTI DDS, all solutions provide their full features sets in an open source distribution. DDS provides a subset of its features under a limited open-source license. Commercial support is available for all of the solutions.

There are a number of performance studies and comparisons available for the solutions surveyed. [11] [12] [13] [14] Specific performance comparisons are difficult to draw without targeted testing accounting for application-specific constraints & requirements regarding scale, reliability, message persistence, transactionality, protocols, etc. In general the solutions surveyed support throughput ranges of several thousand to several million messages per second, depending on a multitude of factors (message size, network & hardware specs, operating system, etc.).

ZeroMQ and DDS are generally considered to be higher performance solutions than the other (brokered) solutions surveyed. Among brokered solutions, RabbitMQ appears to perform particularly well. [11] Targeted performance prototyping has been identified as a high-priority follow-on activity for select messaging solutions in E3.

The relative prevalence of solutions is difficult to measure. Both DDS and AMQP standards have significant commercial backing and large communities. Internet search term frequency analysis – e.g. Google Trends – provides one (coarse) measure of interest in solutions over time. As depicted in Figure 7, RabbitMQ and ActiveMQ currently have significantly higher search term occurrence relative to the other solutions. RabbitMQ has the highest, having passed ActiveMQ in 2013, and shows the most significant growth in recent years among the surveyed solutions.

In summary, AMQP solutions appear to provide a better complexity/feature set trade-off when compared with DDS solutions. Among the AMQP solutions surveyed, RabbitMQ appears particularly compelling, based on prevalence and performance. E3 prototyping for data distribution and processing control will be based on RabbitMQ. This decision will be revisited based on E3 results.

3.2.4. Data Caching & Data Grids

Access to data is a key concern within nearly any application. For data-intensive applications where scalability and response time are significant drivers, timely access to high-volume data is a challenging problem. A number of solutions have evolved in recent years to address this problem, including data caching solutions and data grids.

These solutions are designed to provide low-latency access to both temporary and persistent data across distributed architectures. They are typically designed to integrate with multiple storage solutions.

As described the following sections, there is a wide variety of open-source caching solutions that can be leveraged for applicable processing architectures. Appendix E provides an overview of data caching & data grid architectures, standards and use cases.

3.2.4.1. OSD Data Distribution

As currently defined in the US NDC architecture, the OSD mechanism provides publish/subscribe-based distribution of data across processes, including both persistent data stored in the database, and transient data maintained in memory. Depending on the access performance of the database, a data cache or data grid solution may be used to provide lower-latency access to application data, particularly as the volume of data scales in the future. Figure 4 illustrates an example architecture employing a caching/data grid solution to provide horizontally-scalable application access to system data, both persistent and transient. Although this figure shows a separate messaging framework providing

notification of data updates, many of the surveyed solutions provide a similar capability that could be used instead.

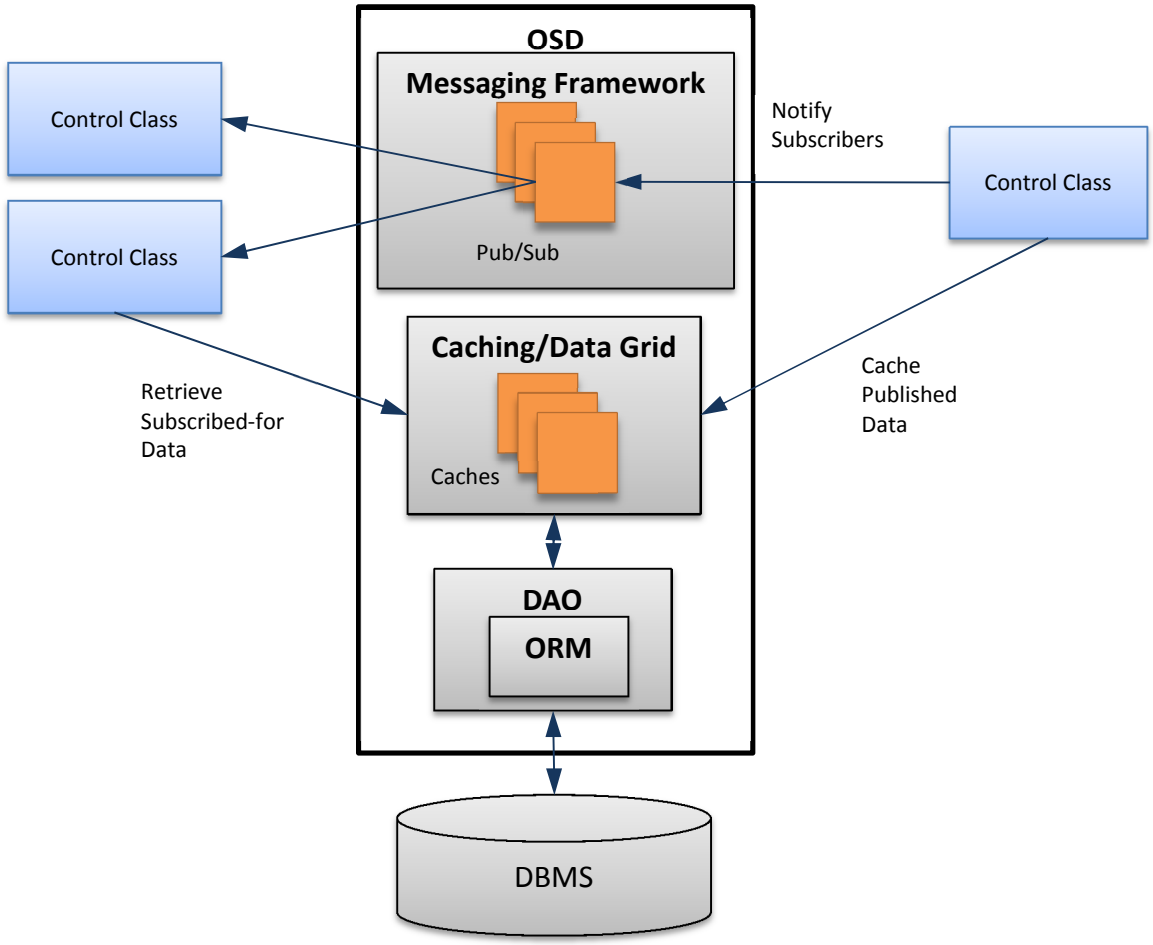

Figure 4. Example Caching As Part of OSD Data Distribution

Application components publish data indirectly to other components by storing data in the cache via Data Access Objects (DAOs) and publishing notification messages to subscribers. Other components subscribe for data through messaging publish/subscribe topics. Upon receipt of notification messages, subscribers retrieve the indicated data from the cache.

3.2.4.2. Survey of Solutions

Several popular, open-source data grid solutions were surveyed as part of the E2 work, including JCS, memcached, EhCache, Infinispan, Redis & Hazelcast. The following sections provide an overview of each framework, as well as high-level

comparisons & conclusions. Table 2 in Appendix D summarizes the survey findings.

3.2.4.2.1. JCS

The Java Caching System (JCS) is a general-purpose open-source caching solution for the Java platform. It is managed by the Apache Software Foundation as part of the Apache Commons project, where it has been under active development since its initial formal release in 2007. It is released under the Apache License 2.0. Although JCS is very similar to the JSR 107 standard, by design it is not a complete implementation of JSR 107[8]. JCS supports four modes [15], addressing both local and distributed caching use cases:

- LRU Memory Cache – An in-memory cache embedded within the application, providing fast access to local data. The cache is managed using an LRU eviction policy.

- Disk Cache – An in-memory embedded cache providing integration with a local backing store. Two types of backing store are provided.

    - The indexed disk cache provides a swapping cache where indices of elements stored on disk are maintained in the cache. It is primarily intended to increase the available cache size beyond the available memory of the application.

    - The JDBC disk cache provides an in-memory cache backed by a relational database. Oracle, MySQL & HSQL are currently supported. Cache elements are serialized and written as BLOBs to the underlying database.

- TCP Lateral Cache - A distributed caching model where each application maintains a local copy of the cache and a connection to every other application (see Figure 16). Cache writes are replicated to all applications participating in the distributed cache. This mode is primarily intended for single-writer topologies, due to the potential for inconsistency across distributed copies (see the Limitations section below).

- RMI Remote Cache - A distributed caching model where each application maintains a connection to a remote caching server, which propagates writes to all of the applications. Multiple cache servers can be configured into a cluster to support fault tolerance and high availability (see Figure

[8] The authors of JCS disagreed with some of the design ideas specified in JSR 107 based primarily on performance concerns.

15). As with the TCP Lateral Caching, this mode does not guarantee consistency given multiple writers.

By default, JCS uses Java's built-in serialization for cache entries; however, custom serializers can be injected in order to optimize serialization performance.

Limitation of JCS include the following:

- It supports only Java applications.
- Its distributed caching modes do not guarantee consistency given multiple writers. Because cache writes are queued, separate writers can overwrite one another's changes in the TCP Lateral cache mode. The RMI remote cache mode makes this situation less likely, as local cache copies are invalidated upon writes to the remote cache server; however the copy stored in the remote server is not guaranteed to be the last created.

3.2.4.2.2. Memcached

Memcached is an open-source, distributed, in-memory key-value store. [16] It was originally developed for LiveJournal in 2003, and is currently maintained by Danga interactive. It is a widely used in high-profile applications such as Youtube, Reddit, Facebook & Twitter, and it is available as part of several prominent cloud computing platforms, including Amazon Web Services (AWS), Microsoft Azure, Google App Engine, Heroku. [17] It is released under the revised BSD License.

Memcached employs a client-server model with the distributed cache partitioned across a set of servers based on key hashing as depicted in Figure 17. This approach provides horizontal scalability insofar as the cache size can be increased through the addition of partition servers. Unlike Figure 17, Memcached does not maintain a partial copy of cache data in the client; all data access is via the partition servers. Cache entries are evicted from the based on the LRU algorithm. Partition servers are not backed to an underlying storage solution.

Memcached is cross-language compatible, and provides clients in many languages, including C, C++, Java, Python, Ruby, Perl & C# among others. In order to provide this support, Memcached does not have built-in serialization for cache entries. Client applications are responsible for data serialization.

Memcached can be configured with authentication using the Simple Authentication and Security Layer (SASL) for deployments on untrusted networks.

The primary limitation of Memcached is that it does not support an underlying data storage solution. Applications are responsible for storage of cached data. As a result, Memcached is not fault tolerant and must be regarded as a strictly transitory cache. A variant of Memcached called MemcacheDB was developed to provide storage of cache data; however it no longer appears to be under active development (the last release was in 2008).

3.2.4.2.3. EhCache/Big Memory

EhCache is an open-source, distributed in-memory key-value store with optional persistence. [18] It has been under active development since 2003, and currently is managed by Terracotta, Inc. EhCache refers to a free, open-source caching solution released under the Apache License 2.0; Terracotta also provides a pair of commercial versions under the names Big Memory Go & Max.

The primary differences between EhCache and the commercial versions relate to cache distribution and cross-language support. EhCache supports standalone mode (single-node cache) and replicated mode (a distributed based on replication). The Big Memory Max cache, in contrast, provides a distributed, caching model combining clustered caching servers and local caches managed based on eviction policies (e.g. LRU, LIRS). Figure 15 & Figure 16 illustrate notional replicated EhCache and Big Memory Max caching models, respectively. EhCache provides only Java-based APIs, with data serialization based on Java serialization. Big Memory Max provides additional clients in C++ & C#, with cross-language data serialization support based on Apache Thrift.

Cached data lifecycle management for both EhCache and Big Memory is based on lifespan policies (Time-to-Idle & Time-to-Live), and Least-Recently-Used (LRU) eviction policies. Cache event callback APIs are provided for client notification of additions, updates, removals, and expirations.

Both EhCache & Big Memory support JTA transactions & optional replication. Both also support persistence of cached data with two separate approaches:

1. Built-in persistence to a custom backing store – this feature appears primarily intended to support durability across application restarts.

2. In addition, EhCache and Big Memory can be configured as either a write-through or write-behind cache for underlying relational databases. In particular, both can be configured as level 2 caches for Hibernate.

EhCache and Big Memory support cache searching based on specially-exposed attributes in either the cached key or value. However, only the Big Memory caches support indexing to improve search performance.

3.2.4.2.4. Infinispan

Infinispan is an open-source distributed in-memory key-value store with optional persistence. [19] [20] It has been under active development since 2009 as a replacement for a previous product called JBoss Cache, and is released under the Apache License 2.0. It is maintained by the JBoss division of RedHat, which provides optional commercial support.

Infinispan supports several deployment models, including:

- **Local** – A non-distributed cache, where data items are stored in the local JVM only. This model is used for basic caching applications where distribution is not required – for example a write-through cache optimizing access to an underlying relational database (see Figure 13).

- **Replicated** – A multi-node cache where the entire cache is replicated to every node. This model leverages local storage to provide faster client access to the entire cache space than the Distributed model; however the cache size is limited to the memory (without a backing store) and disk space (with a backing store) on each node.

- **Distributed** – A multi-node cache where the cache is partitioned across nodes such that each node contains a subset of the key space (see Figure 18). This approach increases network traffic and access latency over the Replicated model; however it provides horizontal scalability of the cache size (memory & disk) through the addition of nodes to the cache.

Both synchronous and asynchronous write propagation models are available depending on the availability and consistency requirements of the application.

Infinispan integrates with numerous storage solutions, including the following:

- Relational databases via Java Database Connectivity (JDBC) & Java Persistence API (JPA)-compliant Object Relational Mapping (ORM) solutions

- NoSQL data stores, including Apache Cassandra, Apache HBase, MongoDB, LevelDB & BerlkeyDB

- Remote Infinispan backing stores via the HotRod protocol (e.g. for distributed caching to a single data store)

- Cloud-based storage services such as Amazon's S3 & Rackspace's CloudFiles, using JCloud

When deployed using the Distributed model, Infinispan supports multi-language applications. This support is implemented as a set of language-specific clients that communicate via either the custom Infinispan protocol called HotRod, or the Memcached wire protocol. Clients are available in C++, Java, Python, Ruby & C#. In order to provide cross-language support, clients are responsible for serialization of data using a cross-language format (e.g. Protocol Buffers, Apache Thrift, Apache Avro, JSON, etc.).

Infinispan provides two search mechanisms. A custom SQL-like, object-based query language is available for both the embedded Java client, as well as remote clients (Java and other languages). A second query interface based on Hibernate Search] and Apache Lucene is available for the embedded Java client only. Both mechanisms include configurable indexing to optimize query performance.

Infinispan provides a number of additional features, including:

- Eviction (LRU, LIRS) & Expiration policies to manage cache memory
- Optional transaction support compliant with the Java Transaction API (JTA) and XA standards.
- Concurrency support using Multiversion concurrency control (MVCC)
- Cache Listeners supporting application processing based on cache events (insertions, updates, deletions)
- Authentication using Java Authentication & Authorization Service (JAAS) and encryption via SSL
- Distributed execution service for cache processing tasks
- Cache management & monitoring support via Java Management Extensions (JMX)
- Command-line Interface
- Integration with Java Enterprise Edition Application Servers (e.g. JBOSS Wildfly)

3.2.4.2.5. Redis

The **Re**mote **Di**ctionary **S**tore (Redis) is an open source, distributed key-value cache with optional persistence to a backing store. [21] Redis was original developed by VMWare, and has been under active development since the early 2000s. It is currently released under the BSD license by Pivotal Software, which provides commercial support. Redis has seen widespread adoption; it was recently ranked as the most popular key-value store by DB-Engines. [22] It is available as a service in a number of cloud environments, including Amazon Web Services S3, Rackspace CloudFiles & Heroku.

Redis employs a cache distribution model based on clustered partition servers with redundant master/slave replication for fault tolerance as depicted in Figure 17. The Cache is *sharded* (partitioned) across the servers, and each shard is replicated to one or more backup servers.

Redis supports two methods for storage of cache elements. The primary data storage method records every command issued to the cache cluster using a set of Append-Only Files (AOF). Upon restart, the log of commands is replayed, recreating the Redis Cache. Alternately, snapshots of the cache partitions can be written to disk based on pre-configured conditions (e.g. elapsed time, accumulated cache changes).

Redis supports clients implemented in numerous development languages, including C, C++, Java, Perl, Python, Ruby, C#, Closure, Scala among others. In order to provide cross-language support, clients are responsible for serialization of data using a cross-language format (e.g. Protocol Buffers, Apache Thrift, Apache Avro, JSON, etc.).

Redis includes a number of additional features, including the following:

- Optional transaction & concurrency support
- Cache Listeners supporting application processing based on cache events (insertions, updates, deletions)
- Eviction (LRU, LIRS) & Expiration policies to manage cache memory
- Message queues & publish/subscribe topics supporting communication across members of the distributed cache.

Limitations of Redis include the following:

- Redis does not provide authentication, encryption or other security features. It is intended to be deployed on a trusted network with an externally-provided security model.

- Redis does not provide facilities for ad-hoc querying of cache entries as other solutions do. Client applications must construct indexes for search applications beyond simple key-based retrieval.

3.2.4.2.6. HazelCast

Hazelcast is a distributed, in-memory & persistent key-values tore for the Java platform. [23] It has been under active development since 2008, and is currently maintained by the company of the same name. Hazelcast is available as an open-source project released under the Apache License 2.0. A commercial version is available as well. Commercial support for the open source version of Hazelcast is available. The commercial version is known as Hazelcast Enterprise, and adds a number of significant features, including:

- Clients for C++ and C# applications

- Encryption, authentication, and other security features

- Off-Heap cache storage using Java NIO to minimize Java garbage collection overhead

- Event-driven continuous queries, supporting Complex Event Processor (CEP) applications

Hazelcast uses a horizontally scalable distributed cache architecture based on partitioning as depicted in Figure 17. In addition to the primary cache data, backup copies can be partitioned across the server cluster providing a degree of fault tolerance. Hazelcast provides numerous additional features, including:

- Support for addition cache data structures, including lists, sets & multi-map

- Message queues & publish/subscribe topics supporting communication across members of the distributed cache.

- Eviction based on LRU and LFU (Least-Frequently Used)

- Write-through and write-behind API for integration with data storage solutions

- Pluggable serializers for performance optimization (beyond built-in Java serialization) and (and cross-language support)
- Support for cache queries using an SQL-like language, as well as indexing for query optimization
- Optional transaction support (using the Entry Processor API)
- Cache Listeners supporting application processing based on cache events (insertions, updates, deletions)

The primary limitation of Hazelcast as an open-source product is that security features and cross-language support are available in the commercial enterprise edition.

3.2.5. Summary

The surveyed solutions generally provide a highly similar set of capabilities – distributed, partitioned data caching with support for queries, transactions, security, data expiration, clients in multiple languages, and integration with underlying storage solution. Commercial support is available for all of the solutions surveyed. Many of the solutions (Redis, Memcached, JCS & Infinispan) offer their full feature sets in a free, open-source version. Others such as EhCache and Hazelcast provide only a limited subset in their free, open-source versions. JCS does not support partitioning for horizontal scalability. Memcached utilizes a client/server model and does not include an in-client near cache for optimized access to commonly needed data.

Among the truly open-source options, Redis appears to provide the most rich feature set. It also appears to be the most popular caching solution among those surveyed (see Figure 12 in Appendix D for the Google Trends graph).

It is not yet clear whether data grids provide a compelling advantage for the system. An analysis of predicted data load and DBMS scaling capacity must be completed in order to determine whether a caching solution or data grid is needed. Pending that analysis, the prototyping work focus on a combination of the database (via DAOs) and messaging software to support data distribution in E3. Should the need arise to incorporate caching the executable architecture prototype the team will start with Redis as the most promising candidate.

4. PROCESSING CONTROL PROTOTYPING

4.1. E1 Background

The E2 OSD prototyping effort represents a progression of the activities completed in E1. A brief summary of the E1 OSD work is provided below for context.

E1 processing control work focused on evaluation of COTS application frameworks supporting development & deployment of a distributed processing architecture. The survey covered a diverse set of technologies, including Java EE application servers, the Spring core framework, stream processors, complex event processors and enterprise service bus (ESB) solutions. Based on the results of the survey, follow-on exploratory prototypes were developed for the Wildfly Java EE application server and the Storm stream processor.

The application frameworks evaluated in E1 were found to be problematic for various reasons. See [24] for specific findings. In response, the team shifted its focus in E2 to simpler frameworks and lower-level technologies that could be used to develop a more tailored custom processing control solution.

4.2. E2 Prototyping

The following activities were completed in E2:

1. Survey of messaging technologies supporting OSD data distribution, as well as processing control software infrastructure
2. Initial research into Java batch processing frameworks

The following sections describe each of these activities in further detail

4.2.1. Messaging Software

Messaging support for the processing control infrastructure was investigated as part of a joint effort addressing both processing control and OSD data distribution. See Section 3.2.3 for results of the E2 of the messaging software survey.

4.2.2. Batch Processing Frameworks

The E1 survey of application frameworks did not identify a particularly compelling candidate to support processing control functions. In E2, the team began to investigate alternative technologies, specifically batch processing frameworks.

Batch processing here refers to the execution of a series of bulk data-oriented application tasks ("jobs") that can be executed to completion without human interaction. [25] It does not imply that processing is executed according to a schedule (although this is one possible approach); nor does it imply that processing cannot include complex structures supporting parallel processing, conditional logic, task decomposition, etc. Under this broad definition, both the automated processing and interactive Analyst support functions can be thought of as batch processing jobs.

Two prominent Java frameworks have emerged in recent years to support batch processing: Spring Batch and the Java EE Batch (JSR 352). Spring Batch was first released in 2007, and has seen a steady increase in interest since (as indicated by Google Trends). In 2013, JSR 352 was released, specifying batch processing support for Java EE application servers. The framework spelled out in JSR 352 is strikingly similar to the Spring Batch framework, and highlights the fact that Spring Batch has served as the de facto standard. JSR 352 is now supported by most Java EE application servers, including Widlfly, Glassfish, Websphere and WebLogic.

Both frameworks provide essentially the same basic components and patterns for composing batch processing applications. Figure 5 illustrates a sample batch processing application. As shown in the figure, data processing is organized into jobs, which consist of steps arranged in sequences, parallel processing paths and conditional flows. The structure of a job is defined using a declarative specification, either using XML or in-code annotations. The Job Launcher orchestrates execution of the jobs at runtime. Information about jobs is stored in the job repository.

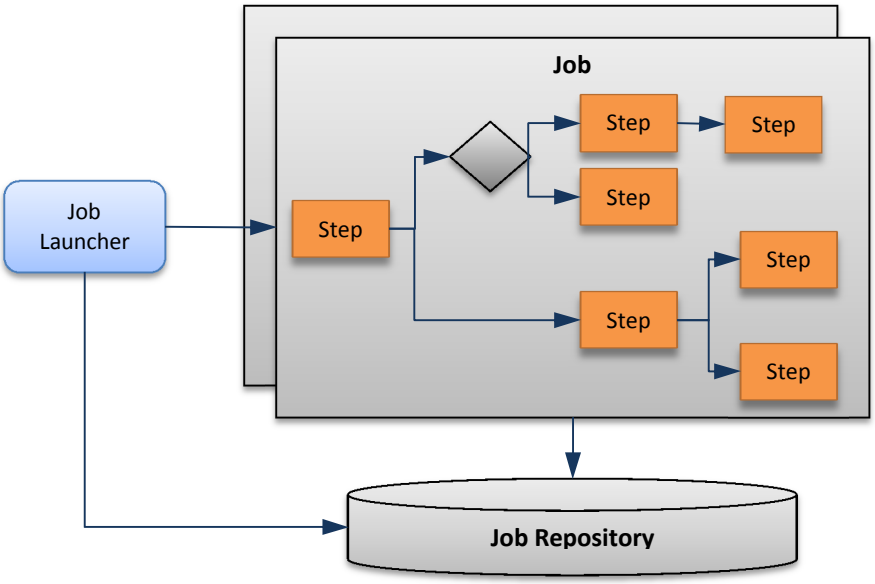

Figure 5. Example Batch Processing Application

A Job Launcher (Job Operator in JSR 352) manages the execution of jobs in the application, configuring, starting, stopping, retrying, etc. jobs based on rules defined in the Job specification (XML or annotations). Information about current and past jobs is stored in a job repository. Jobs consist of steps, which can be arranged in sequences, in parallel, and based on conditional logic as defined in the Job Specification.

Both frameworks provide the following:

- Job definition language
- Runtime environment for execution of batch jobs
- APIs to define jobs, steps, conditional logic, etc.
- Transaction support
- Error handling and recovery (e.g. job retry)
- Management of state information for jobs and steps

A significant feature distinguishing Spring Batch from JSR 352 is that Spring Batch supports federation of job execution across multiple processes, with message-based IPC supporting the coordination of individual steps. In contrast,

JSR 352 supports only multi-threaded job execution within a single JVM. Support for multi-process job execution makes Spring Batch particularly attractive.

Overall, Spring Batch appears to align best with the system processing control infrastructure requirements defined in Section 2.2. However, the initial investigation completed in E2 is incomplete and requires follow on work to further assess its suitability for the system. Example questions that must be addressed include:

- What (if any) support is available for steps implemented in languages other than Java (e.g. C++)?
- What performance implications do these frameworks have for the applications they support?

5. PATH FORWARD

Starting in E3, the team will begin design and development work directly addressing the executable architecture deliverable. Moving forward, the prototyping team will expand to support both the executable architecture development effort, as well as follow exploratory prototyping in support of architecture definition. An initial plan and schedule for development of the executable architecture is documented in [26]. Specific E3 activities that will be addressed as part of, or in addition to, the work documented in this plan include the following.

5.1. OSD

- Performance assessment of the Hibernate ORM based on worst-case scenarios.
- Selection of cross-language serialization COTS supporting direct data distribution between processing components implemented in different languages (nominally Java and C/C++).[9]

[9] In E2, the prototyping team completed a preliminary investigation of cross-language serialization COTS, including Google protocol buffers, Apache Thrift, Apache Avro, Splice and JSON. Google protocol buffers and Apache Thrift were identified as the most promising solutions, based on serialization performance, popularity and maturity. However a decision was not reached as to which of the two should be used in development of the executable architecture prototype moving forward.

5.2. Processing Control

- Follow-on investigation and prototyping of batch processing frameworks supporting the processing control infrastructure

6. WORKS CITED

[1] Sandia National Laboratories, "US NDC Modernization Iteration E2 Prototyping Report: User Interface Framework," 2014.

[2] "JPype 0.4 - User Guide," [Online]. Available: http://jpype.sourceforge.net/doc/user-guide/userguide.html. [Accessed 31 August 2014].

[3] "ActiveMQ Support," Apache Software Foundation, [Online]. Available: http://activemq.apache.org/support.html. [Accessed 20 August 2014].

[4] "ActiveMQ," Apache Software Foundation, 18 July 2014. [Online]. Available: http://activemq.apache.org/. [Accessed 19 July 2014].

[5] "Apollo, ActiveMQ's Next Generation of Messaging," Apache Software Foundation, [Online]. Available: http://activemq.apache.org/apollo/index.html. [Accessed 19 July 2014].

[6] H. Chirino, "Stomp Server Performance Comparisons," FuseSource, [Online]. Available: http://hiramchirino.com/stomp-benchmark/ec2-c1.xlarge/index.html. [Accessed 19 July 2014].

[7] "øMQ - The Guide," iMatix Corporation, [Online]. Available: http://zguide.zeromq.org/page:all. [Accessed 31 August 2014].

[8] "Real-Time Innovations, Inc. Open Infrastructure Community License," Real-Time Innovations, [Online]. Available: https://www.rti.com/downloads/IC-license.html. [Accessed 14 August 2014].

[9] "Leadership Through Connext DDS," Real-Time Innovations, Inc., [Online]. Available: http://www.rti.com/products/dds/dds_leader.html. [Accessed 14 August 2014].

[10] "Solving Customer Problems," Real-Time Innovations, Inc., [Online]. Available: http://www.rti.com/industries/index.html. [Accessed 14 August 2014].

[11] M. Salvan, "A quick message queue benchmark: ActiveMQ, RabbitMQ, HornetQ, QPID, Apollo…," Muriel's Tech Blog, 10 April 2013. [Online]. Available: http://blog.x-aeon.com/2013/04/10/a-quick-message-queue-benchmark-activemq-rabbitmq-hornetq-qpid-apollo/. [Accessed 15 July 2014].

[12] "RabbitMQ Performance Measurements, part 2," Pivotal Software, Inc., [Online]. Available:

http://www.rabbitmq.com/blog/2012/04/25/rabbitmq-performance-measurements-part-2/. [Accessed 6 August 2014].

[13] "ZeroMQ Performance Tests," iMatix Corporation, 13 April 2011. [Online]. Available: http://zeromq.org/area:results. [Accessed 5 August 2014].

[14] "Apache QPid Performance," Apache Software Foundation, [Online]. Available: http://qpid.apache.org/releases/qpid-0.20/java-broker/book/Java-Broker-High-Availability-Performance.html. [Accessed 5 August 2014].

[15] "Java Caching System," Apache Software Foundation, 25 March 2014. [Online]. Available: http://commons.apache.org/proper/commons-jcs. [Accessed 8 August 2014].

[16] "memcached Wiki," 22 August 2011. [Online]. Available: https://code.google.com/p/memcached/wiki/NewOverview. [Accessed 8 August 2014].

[17] "Memcached," Wikiepdia, 14 July 2014. [Online]. Available: http://en.wikipedia.org/wiki/Memcached. [Accessed 8 August 2014].

[18] D. Wind, Instant Effective Caching with EhCache, Birmingham, UK: Packt Publishing, 2013.

[19] M. M. G. Z. P. M. Manik Surtani, "Infinispan User Guide," Red Hat, Inc., 11 August 2014. [Online]. Available: http://infinispan.org/docs/6.0.x/user_guide/user_guide.html. [Accessed 12 August 2014].

[20] F. &. S. M. Marchioni, Infinispan Data Grid Platform, Birmingham, UK: Packt Publishing, 2012.

[21] "Redis Documentation," Pivotal, [Online]. Available: http://redis.io/documentation. [Accessed 5 August 2014].

[22] "DB-Engines Rankin gof Key Value Stores," Solid IT, August 2014. [Online]. Available: http://db-engines.com/en/ranking/key-value+store. [Accessed 14 August 2014].

[23] M. Johns, Getting Started with Hazelcast, Birmingham, UK: Packt Publishing, 2013.

[24] Sandia National Laboratories, "US NDC Modernization Iteration E1 Prototyping Report: Processing Control Framework," 2014.

[25] "Batch Processing," Wikipedia, 15 September 2014. [Online]. Available: http://en.wikipedia.org/wiki/Batch_processing. [Accessed 21 September 2014].

[26] Sandia National Laboratories, "US NDC Modernization Iteration E2 Prototyping Report:

Executable Architecture Planning".

[27] "OASIS Advanced Meesage Queueing Protocol (AMQP) Version 1.0," 29 October 2012. [Online]. Available: http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-overview-v1.0-os.html. [Accessed 3 July 2014].

[28] "Advanced Message Queueing Protocol," Wikipedia, 24 June 2014. [Online]. Available: http://en.wikipedia.org/wiki/AMQP. [Accessed 5 July 2014].

[29] "STOMP Protocol Specification, Version 1.2," 22 October 2012. [Online]. Available: http://stomp.github.io/. [Accessed 5 July 2014].

[30] v. Team, "Choosing Your Messaging Protocol: AMQP, MQTT, or STOMP," VMWare BLOGS, 19 February 2013. [Online]. Available: http://blogs.vmware.com/vfabric/2013/02/choosing-your-messaging-protocol-amqp-mqtt-or-stomp.html. [Accessed 5 July 2014].

[31] "Data Distribution Service for Real-time Systems Version 1.2," Object Management Group, 2007.

[32] "Data Distribution Service," Wikipedia, 13 July 2014. [Online]. Available: http://en.wikipedia.org/wiki/Data_Distribution_Service. [Accessed 16 July 2014].

[33] G. Pardo-Castellote, "RTI DDS: A Next-Generation Approach to Building Distributed Real-Time Systems," Real-Time Innovations, 25 February 2011. [Online]. Available: http://www.slideshare.net/GerardoPardo/rti-datadistribution-service-dds-master-class-2011. [Accessed 14 August 2014].

[34] "JSR 107: JCache - Java Temporary Caching API," Oracle Corporation, [Online]. Available: https://jcp.org/en/jsr/detail?id=107. [Accessed 20 June 2015].

[35] "JSR 347: Data Grids for the Java Platform," Oracle Corporation, [Online]. Available: https://jcp.org/en/jsr/detail?id=347. [Accessed 20 June 2014].

[36] Sandia National Laboratories, "US NDC Modernization Iteration E1 Prototyping Report: Processing Control Framework," 2014.

APPENDIX A. OSD OVERVIEW

As the architecture definition effort expanded in E2, the project team developed preliminary definitions for a number of key mechanisms, including the OSD. Figure 6 illustrates the OSD as defined in E2, highlighting prototyping areas of focus. As depicted in the figure, the OSD is a software mechanism consisting of two primary elements: *Stored Data Access* & *Data Distribution*.

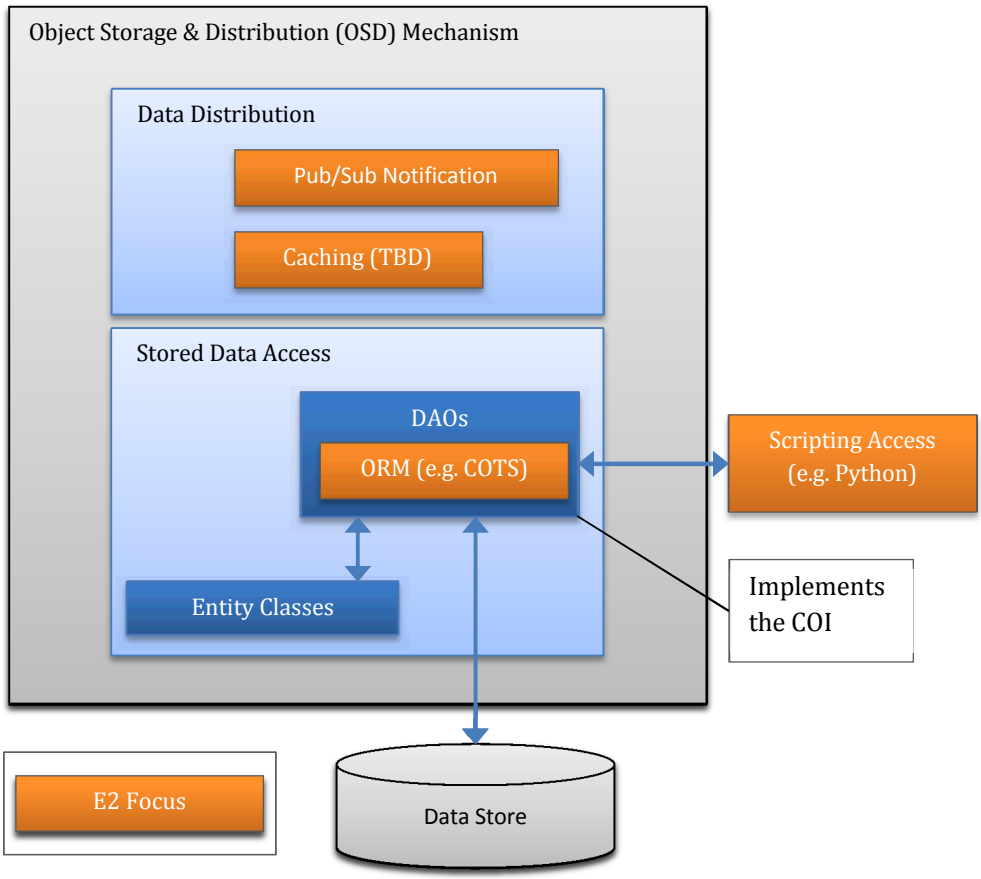

Figure 6. Object Storage & Distribution Mechanism as of E2

A.1. OSD Stored Data Access

Stored Data Access provides the system with an object-based CRUD[10] interface to data stored in the underlying database management system (DBMS). Access is provided via a set of entity classes encapsulating system data, and a set of Data Access Objects (DAOs) providing APIs for entity-based access to the database. The DAOs provide an implementation of the *Common Object Interface* (COI), which specifies a language-agnostic and DBMS-agnostic set of interfaces for access to system data.

DAOs are built using Object-Relational Mappings (ORMs), which translate between entities in the application and relational tables in the underlying database.

A.2. OSD Data Distribution

Data Distribution provides access to application data across processes, including both persistent data stored in the database as well as transient data maintained in memory. Distribution is provided through publish/subscribe middleware and standard data serialization technologies. In order to support the full set of use cases, multiple data distribution design patterns will likely be needed. Examples include direct transmission of application data between processes via messaging, and indirect distribution of data via the database or distributed cache

[10] Create, Read, Update & Delete

APPENDIX B. MESSAGING SOLUTION SPACE SUMMARY

Table 1. Summary Comparison of Surveyed Messaging Solutions

Name	Standards	Language Support	Advantages	Disadvantages
RTI DDS	DDS JMS REST SOAP	C, C++ C# Java Ada	• Standards-Based • Cross-Language Support • Designed for low-latency, high-throughput with configurable QoS • Flexible communication patterns & configurable transports • Open-source version available with commercial support from RTI • Generally considered to be higher performance than brokered solutions	• Open-source license is more restrictive than for other solutions • Many features are only available in the commercial edition • Appears to be less popular than other solutions (based on Google Trends) • Configurable QoS introduces complexity relative to other solutions • Past prototyping efforts have struggled with product complexity
Qpid	AMQP JMS	Java C, C++ C# Ruby Perl Python PHP	• Standards-Based • Cross-Language Support • Free OSS with community support	• Appears to be less popular than other solutions (based on Google Trends)
ActiveMQ / Apollo	AMQP STOMP REST XMPP JMS 1.1	Java C, C++ C# Ruby Perl Python PHP	• Standards-Based • Cross-Language Support • Free OSS with community support • Mature & highly stable (widely used since early 2000s) • Highly popular	• Performance limitations at scale (Apollo subproject attempts to address these, but is not yet a full-featured product) • Interest in ActiveMQ appears to be declining in recent years (based on Google trends)

RabbitMQ	AMQP STOMP	Java C++ .NET Ruby Perl Python PHP	• Standards-Based • Cross-Language Support • Free OSS with community support • Commercial support available from Pivotal • Highly popular (highest search term frequency on Google Trends) • Favorable performance on a number of benchmarks	• Broker is implemented in Erlang, which is a relatively obscure development language
ZeroMQ	None	Java C, C++ C# Ruby Perl Python PHP	• Cross-Language Support • Free OSS with community support • Generally considered to be higher performance than brokered solutions	• Not standards-based • Appears to be less popular than other solutions (based on Google Trends)

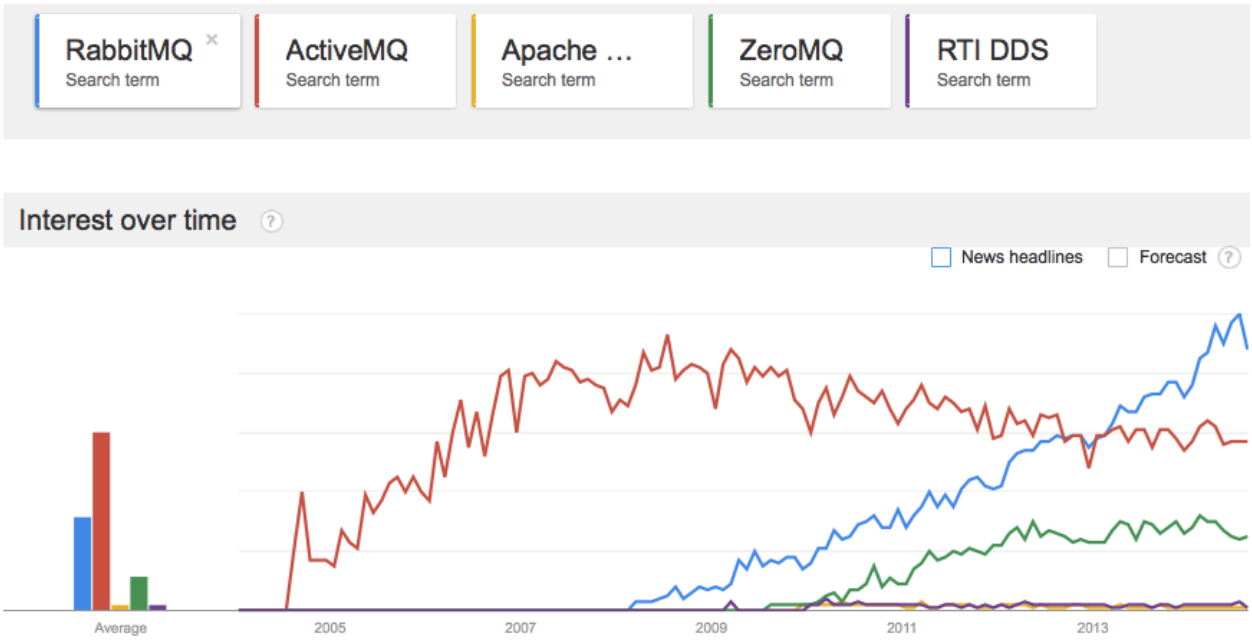

Figure 7. Google Trends Interest over Time for Surveyed Solutions

APPENDIX C. MESSAGING OVERVIEW

The following sections provide an overview of the concepts, standards and use cases related to messaging software.

C.1. Concepts

C.1.1. Message Oriented Middleware

Message-oriented middleware (MOM) is a subset of messaging technology that is designed to provide loosely-coupled, asynchronous, resilient communication between heterogeneous components across platforms and development languages. Most of the messaging solutions included in the survey can be categorized as message-oriented middleware.

C.1.2. Loose Coupling

MOM solutions provide loose coupling between communicating components insofar as the sender and receiver are not directly aware of one another and

communicate asynchronously. Rather than explicitly addressing messages to one or more receivers, the sender addresses messages to a *destination* managed by the messaging provider. The destination may, for example, represent a message queue, or a topic exchange in the case of the publish/subscribe messaging. Communication is asynchronous in that the sender and receiver need not be active or even exist at the same time. The messaging provider stores messages from the sender until the recipient retrieves them. Messaging providers typically implement this capability using message queues.

C.1.3. Resilience

MOM solutions are typically designed to provide resilience in the face of network failures as well as client and provider failures (software, hardware). Most designs employ redundant providers with replicated message stores (e.g. queues) and message acknowledgment strategies to provide resilience through retransmission of messages under failure conditions. Resilience is often expressed in terms of message delivery guarantees.

- *At-most-once* message delivery guarantees that each message will be delivered once or not at all. This guarantee is characteristic of unreliable messaging systems where messages may be lost.

- *At-least-once* message delivery guarantees that each message will be delivered at least once, and may be delivered multiple times. This guarantee is characteristic of reliable messaging systems, typically based on message acknowledgement and retransmission under failure conditions[11].

- *Exactly-once* message delivery guarantees that each message will be delivered once and only once. This is typical of reliable messaging systems that build on the retransmission model used for at-least-once guarantees, adding message deduplication at the receiving end.

Most of the solutions surveyed provide at-least-once delivery guarantees and leave it to the client application to implement exactly-once guarantees, either through idempotent message handling, or message deduplication.

[11] Acknowledgment in this context indicates both that the message has been received and has been acted upon by the client. This is in addition to network layer retransmission, for example using TCP, where retransmission is used to ensure receipt of all packets.

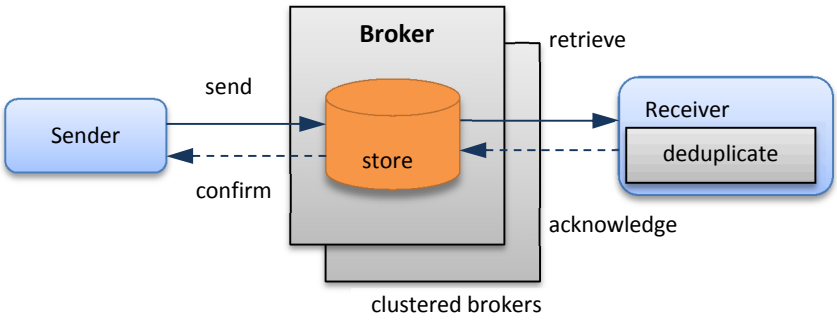

Figure 8. Messaging Resilience Approaches

The broker maintains a persistent cache (e.g. message queue) to support retransmission of messages under failure conditions. Brokers may be clustered to address broker failures. Receivers acknowledge receipt and processing of messages to the broker. Only when the acknowledgement is received will the broker remove messages from the cache. If the acknowledgement message from the receiver to the broker is lost due to a network failure, the broker will continue to retransmit the message. In this case, the receiver must discard subsequent message copies to achieve exactly-once processing guarantees.

C.1.4. Messaging Patterns

MOM solutions support a wide variety of messaging patterns, including for example point-to-point, publish/subscribe, & request/response. Both synchronous and asynchronous messaging are typically supported. In the case of synchronous messaging, the sender will block until the broker confirms receipt of the message by the receiver.

- *Point-to-Point*

  This basic pattern is typically implemented using message queues. As depicted in Figure 9, senders address each message to a specific message queue that is associated with one or more receivers. Once the message is retrieved from the queue and acknowledged by a receiver, it is removed from the queue; thus only one receiver will process each message. This pattern is often used to implement task queues, where workers retrieve task messages from the queue.

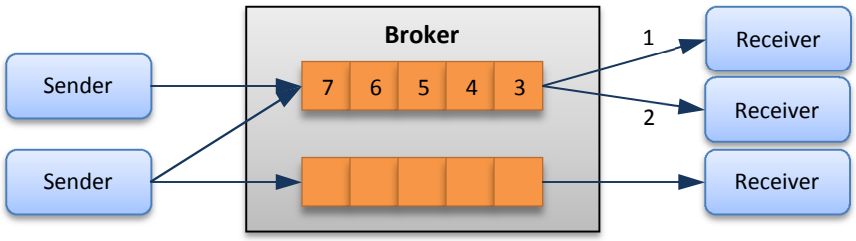

Figure 9. Basic Message Queue Example

One or more senders address messages to a message queue managed by middleware broker(s). One more receivers retrieve the messages from the queue, with each message retrieved by only one receiver.

- *Publish/Subscribe*

  In a publish/subscribe application, senders publish messages to a named topic that serves as a routing key for messages. As depicted in Figure 10, all receivers subscribed to the topic will receive copies of the message. Some solutions provide *durable* subscriptions, where new subscribers receive copies of durable messages sent before the subscription was created.

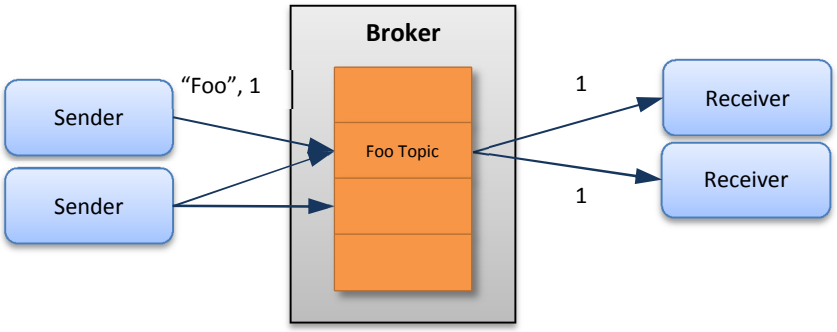

Figure 10. Publish/Subscribe Messaging Example

- *Request/Response*

  In the request/response pattern, the consumer sends a message to a dedicated request queue. The message includes both the request information, as well as the address of a separate queue that will be used to respond to the request. Upon receipt and processing of the request, the producer sends a response message to the specified response queue. This pattern is typically used to implement remote procedure call (RPC) like interactions between processing components. See Figure 11.

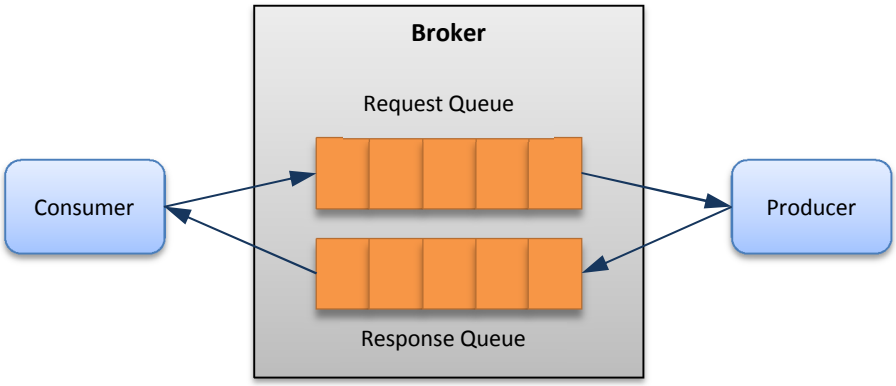

Figure 11. Request/Response Messaging Example

As an extension of the messaging patterns described above, many of the solutions surveyed include additional message filtering capabilities, based on inspection of message content.

C.1.5. Transactions

Many of the surveyed messaging solutions support transactions of some kind. Typically, messages transfers (including acknowledgements) can be grouped into transactional sessions such that the messages only become available at the target address upon commit, thus guaranteeing all-or-none delivery.

C.1.6. Security

The messaging solutions surveyed typically implement some combination of the following security features.

- Authorization using *Access Control Lists* for message brokers
- Authentication of client connections to the message brokers using, for example LDAP or Kerberos
- Encryption and certificate management, typically using SSL

C.1.7. Cross-Language Support

All of the solutions surveyed provide some level of support for clients implemented in multiple languages. Support typically includes a language-agnostic wire protocol and separate client APIs implemented in a variety of languages – typically at least C, C++, Java, Python & C#. For the most part, these solutions do not address serialization of message data into a cross-language compatible format. This task is left to the client application.

C.2. Relevant Standards

Several prominent messaging standards have evolved in recent years that define the communication patterns, protocols & application programming interfaces (APIs) commonly used in distributed applications. These are discussed briefly below.

C.2.1. AMQP

The Advanced Message Queuing Protocol (AMQP) is an open standard for reliable, secure, binary message-based communication. [27] It supports both point-to-point and publish/subscribe communication patterns with multiple reliable messaging policies, including *at-most-once*, *at-least-once* and *exactly-once*[12] delivery guarantees. AMQP Assumes a reliable transport protocol such as TCP. *At-least-once* and *exactly-once* guarantees further depend on message durability[13] and transaction support.

AMQP defines message formats and transfer protocols that enable interoperability across implementations, platforms and development languages. Authentication and encryption support are based on Transport Layer Security (TLS) and Simple Authentication & Security Layer (SASL).

AMQP messaging is asynchronous in that the sender and receiver need not be available at the same time to communicate. It enables loose coupling of application components in that senders and receivers need not know about one another.

AMQP has been under development since 2003. Initial development of the standard was led by JP Morgan Chase, which formed an open working group that grew to include a number of prominent companies in the banking and software industries.[14] The current version of AMQP is 1.0, which was released as an ISO international standard in April of 2014. [28]

C.2.2. STOMP

The Simple/Streaming Text Oriented Message Protocol (STOMP) is a lightweight, open standard for message-based communication. [29] A primary distinguishing feature of STOMP is that messages are text-based, rather than binary, making it somewhat similar to HTTP. Simplicity and interoperability are core design

[12] *At-most-once* delivery guarantees that each message will be delivered once or not at all. *At-least-once* delivery guarantees that each message will be delivered at least once, and may be delivered multiple times. *Exactly-once* delivery guarantees that each message will be delivered once and only once.
[13] Message durability enables retransmission of persisted messages in the case of delivery failures.
[14] Members of the AMQP working group included Bank of America, Barclays, Cisco Systems, Credit Suisse, Goldman Sachs, JPMorgan Chase, Microsoft, Novell, VMWare & RedHat, among others. [28]

principles of the standard. Although STOMP specifies a message format and transfer protocol enabling interoperability, it is less explicit than AMQP on details of the protocol, with the result that broker implementations vary somewhat in practice. [30] Several of the brokers implementing the AMQP standards also provide support for STOMP (see Section 3.2.3.1).

C.2.3. JMS

Java Message Service (JMS) is an API specification for asynchronous, reliable messaging within Java Enterprise Edition (Java EE) applications. Unlike AMQP, JMS does not include message format and transfer protocol definitions; it specifies only the messaging API available to client Java EE applications. As such, it does not enable interoperability across JMS implementations. As a Java standard, it is not intended to enable interoperability across languages.

JMS supports queuing and publish/subscribe messaging patterns with multiple reliable messaging policies, including *at-most-once*, *at-least-once* and *exactly-once* delivery guarantees.

C.2.4. DDS

The Data Distribution Service for Real-Time Systems (DDS) is a standard for scalable, reliable, high-performance communication. [31] DDS specifies a data-centric publish/subscribe communication framework, including wire protocol, architecture and client APIs. It also specifies a data persistence framework.

A significant feature of DDS is its fine-grained, highly configurable control over Quality of Service (QoS) policies. These policies define the behavior of data exchange between sender and receiver related to latency, durability, reliability, sample rate, data partitioning, transport priority, data lifespan, and resource utilization limits, among others.

DDS was developed through a partnership between two vendors: Real-Time Innovations (RTI) and the Thales Group. [32] It was first released as an Object Management Group (OMG) standard in 2003. The current version is 1.2, which was released in 2007. DDS has seen widespread adoption among DoD programs for real-time systems. Examples include the US Navy Open Architecture program, Space and Naval Warfare Systems Command (SPAWAR) Net-centric Enterprise Solutions for Interoperability (NESI) program, and the DoD Information Technology Standards Registry (DISR). [33]

APPENDIX D. CACHING SOLUTION SPACE SUMMARY

Table 2. Summary Comparison of Surveyed Caching Solutions

Name	Client Language Support	Advantages	Disadvantages
JCS	Java	• Cross-Language Support • Free OSS with community support	• Java only (no cross-language support) • Appears to be less widely used/popular than other solutions (e.g. Redis, memcached) • It is not clear whether commercial support is available • Limited feature set relative to other solutions surveyed • Does not support partitioning (only replication)
memcached	C, C++ Java, Python Ruby Perl C#	• Well established and mature • Widely used highly popular • Cross-Language Support • Free OSS with community support • Commercial support available	• Popularity appears to be declining (based on Google Trends)
EHCache	Java C++ &C# (commercial version)	• Cross-Language Support • Free OSS version available • Commercial support available from Terracotta • Strong feature set, including partitioning, replication, transactions, security, etc.	• Many features are only available in the commercial edition • Limited cross-language support (and only in the commercial edition) • Appears to be less widely used/popular than other solutions (e.g. Redis, memcached) • Popularity appears to be declining (based on Google Trends)

Infinispan	C++ Java Python Ruby C#	• Cross-Language Support • Free OSS with community support • Commercial support available from JBoss • Strong feature set, including partitioning, replication, transactions, security, etc.	• Appears to be less widely used/popular than other solutions (e.g. Redis, memcached)
Redis	C, C++ Java Perl Python Ruby C# Closure Scala	• Widely used highly popular • Broad cross-Language Support • Free OSS with community support • Commercial support available from Pivotal • Strong feature set, including partitioning, replication, transactions, etc.	• Limited built-in security features
Hazelcast	Java C++ &C# (commercial version)	• Cross-Language Support • Commercial support available from Hazelcast • Strong feature set, including partitioning, replication, transactions, security, etc.	• Many features are only available in the commercial edition • Limited cross-language support (and only in the commercial edition) • Appears to be less widely used/popular than other solutions (e.g. Redis, memcached)

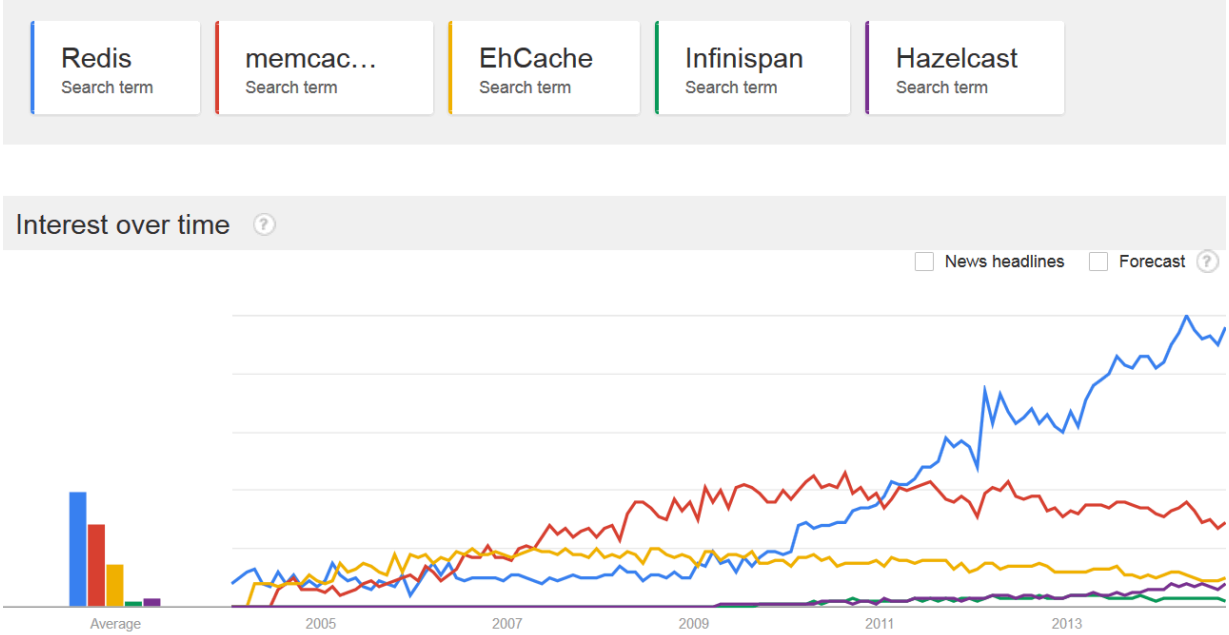

Figure 12. Google Trends Interest over Time for Surveyed Solutions

APPENDIX E. OVERVIEW OF DISTRIBUTED CACHING ARCHITECTURES

E.1. Concepts

E.1.1. Data Cache

In its simplest form, a data caching solutions provides fast access to data – either transient or persistent - by maintaining cached copies of frequently accessed data, typically in a key-value map structure. The cache may be embedded in the application or managed as a separate service. Figure 13 illustrates the basic embedded caching model.

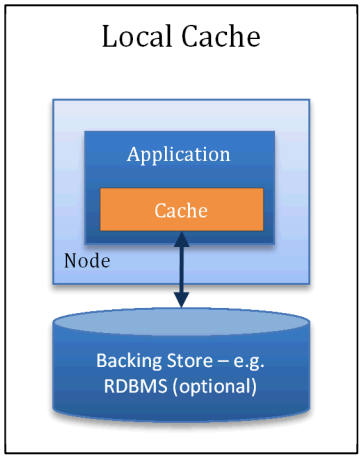

Figure 13. Basic Caching Example

For cases where the cache is integrated with an underlying storage solution, synchronous and asynchronous persistence models are typically available, trading write performance for consistency timeliness guarantees.

- Write-through caching (synchronous) – The cache acts as a façade for the underlying storage solution. Writes to the cache synchronously update the backing store. This approach maintains consistency between the cache and backing store at the expense of write performance.

- Write-behind caching (asynchronous) - The cache acts as a façade for the underlying storage solution. Writes to the cache and backing store are executed asynchronously. This approach minimizes application write time at the expense of temporary inconsistency between the cache and backing store.

The size of the cache is typically managed through a configurable set of policies, including the following.

- Eviction

  The cache is configured with a maximum number of data items. When the maximum is reached, elements are removed from the cache using algorithms designed to identify elements that are less likely to be accessed in the future. Examples include the Least Recently Used (LRU) algorithm and Low Inter-reference Recency Set (LIRS) algorithm.

  - LRU

Cache items are identified for expiration based on elapsed time since the last access request. The underlying assumption of LRU is that the least recently accessed cache items are least likely to be accessed in the future.

- LIRS

  The LIRS algorithm identifies cache items for expiration based on *reuse distance* – the count of cache items accessed since the last request for the cache item in question. LIRS provides improved performance over LRU for cases where the cached data exhibit weak temporal access locality (i.e. where the LRU assumption does not hold).

- Expiration

  Cache entries are configured with a fixed cache lifespan (referred to as Time-to-Live, or TTL) or maximum idle time (referred to as Time-to-Idle, or TTI). When cached items reach the end of their lifespan or maximum idle time, they are removed from the cache.

Caching solutions typically support both key-based search as well as queries based on pre-built indexes configured as part of the cache deployment.

E.1.2. Distributed Caching

A distributed cache builds on the basic concepts described above, adding support for distribution of cached data across applications & processing nodes. This type of solution introduces support for distributed processing architectures and provides varying degrees horizontal scalability not available in the basic caching solution.

Figure 14 shows a basic distributed caching architecture where replicated copies of the cache are embedded within each application. Replication is used to maintain consistency between the caches and can be configured to optimize for consistency or availability based on the needs of the application. Synchronous replication ensures consistency across the distributed cache at the expense of availability; updated data will not be available until replication is complete. Asynchronous replication allows access to all data during replication at the expense of consistency. Until replication is complete, accessed data is not guaranteed to be consistent across the distributed cache.

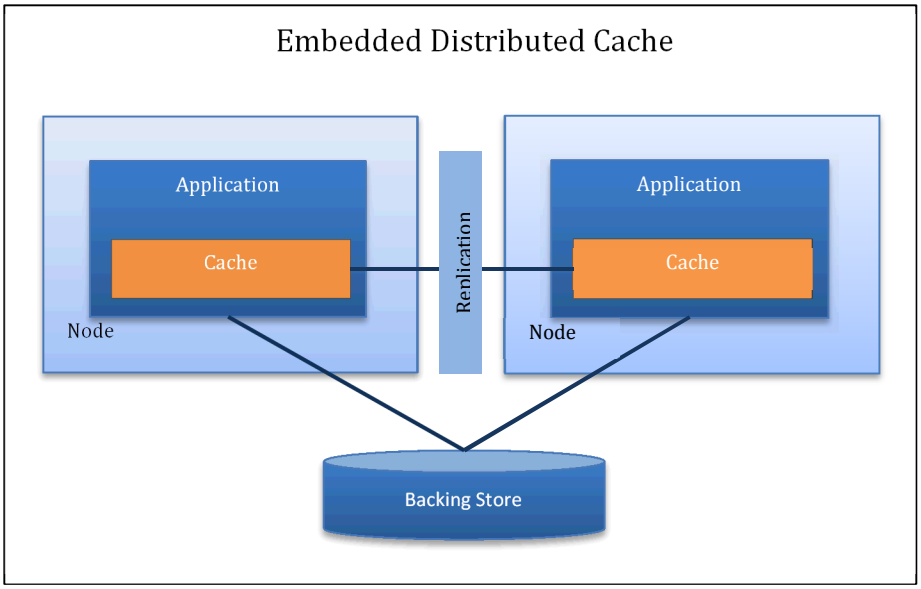

Figure 14. Basic Distributed Caching Example

An alternate distributed caching model establishes a dedicated cluster of caching servers. As shown in Figure 15, applications maintain a *near cache* copy of frequently accessed cache data within the local address space that is replicated, either synchronously or asynchronously with a cluster of dedicated caching servers, each hosting a replicated copy of the full cache. Both the local near cache and full cache instances are managed using expiration and eviction policies.

An advantage of this approach is that it allows for specialization of the hardware within the architecture; applications may be run on lower memory machines (e.g. workstations), while the caching server nodes may be tailored to maximize cache size and responsiveness. Nonetheless, cache size is still limited to the available memory size of the cache servers, and so does not provide horizontal scalability.

A limitation of fully-replicated caching models (Figure 14 & Figure 15) is that the cache cannot scale beyond the available memory of any single node because each node maintains a complete copy.

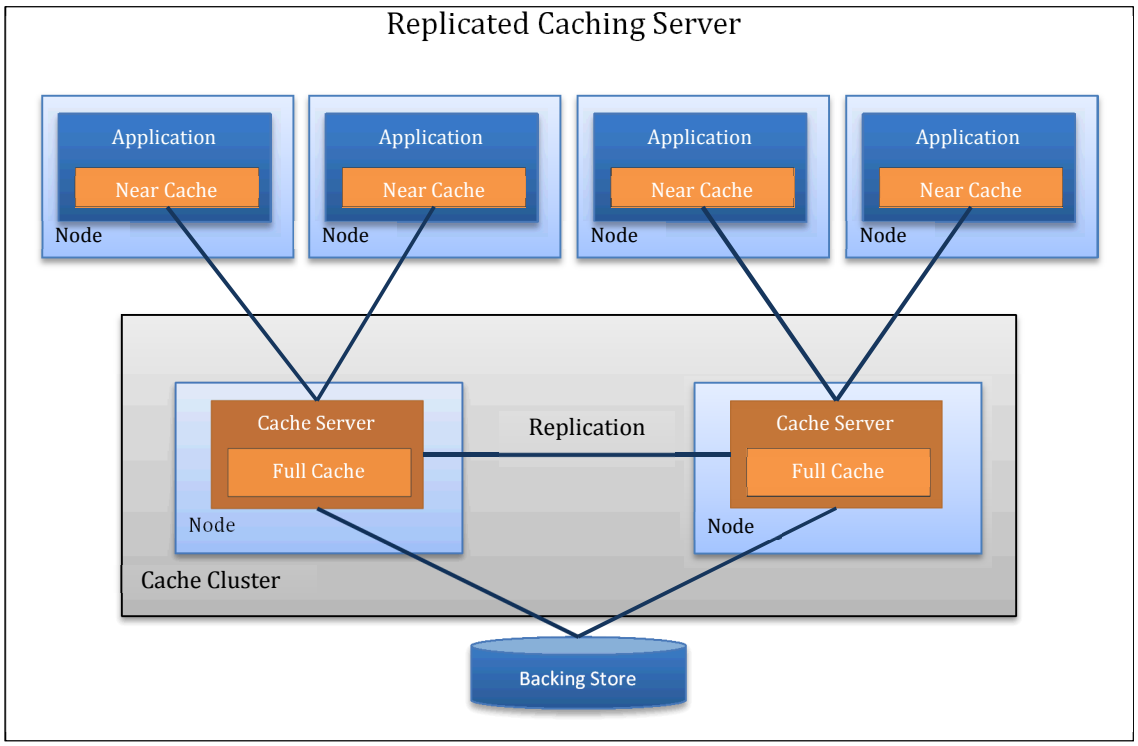

Figure 15. Distributed Caching with Replicated Servers

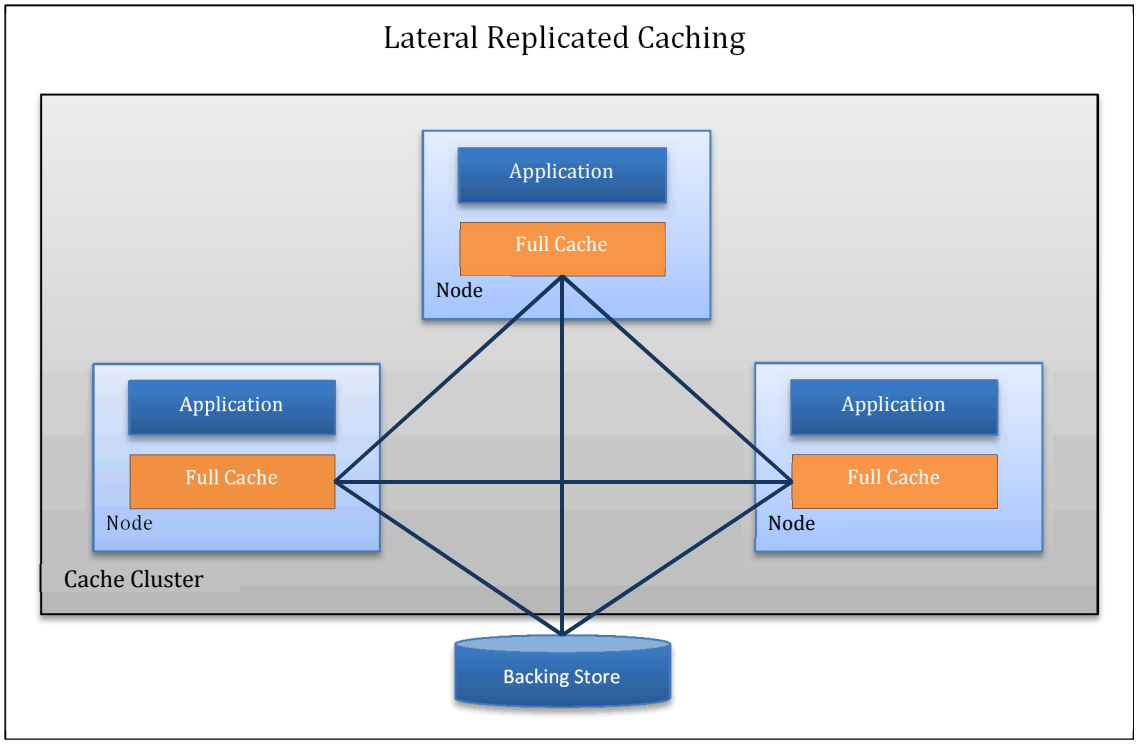

Figure 16. Distributed Caching with Lateral Replication

Horizontal scalability is typically achieved using a partitioned caching model where each cache node hosts a subset of the cache entries, most commonly based on a hash of the key set. Increases in cache size are accommodated by adding nodes to the cache and redistributing the key space. As with other distributed caching architectures, client applications may maintain a near cache copy of frequently accessed cache entries. As illustrated in Figure 17 & Figure 18, partitioned caches can be implemented using either a client-server architecture (see Figure 17), or a lateral model where client application nodes host the cache partitions (see Figure 18).

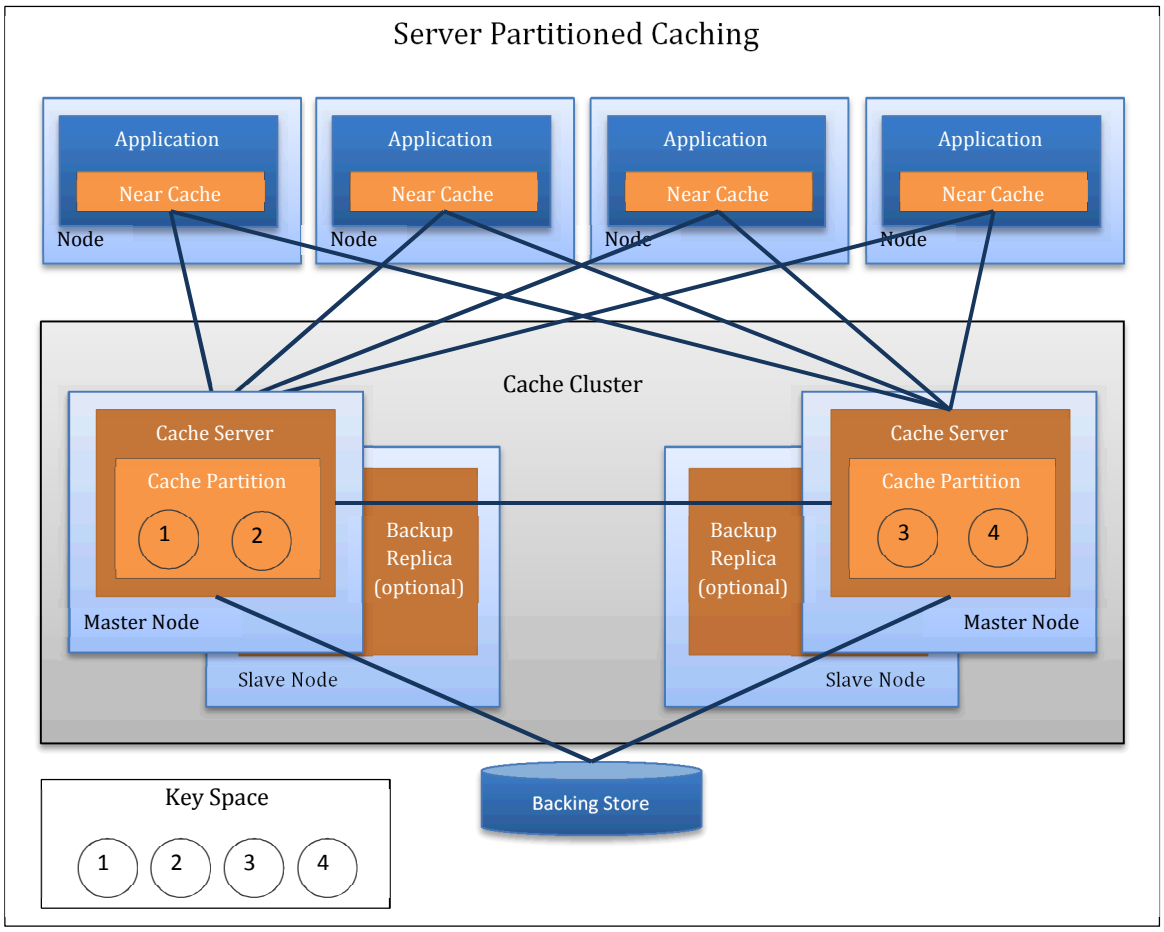

Figure 17. Scalable Distributed Caching with Partitioned Servers

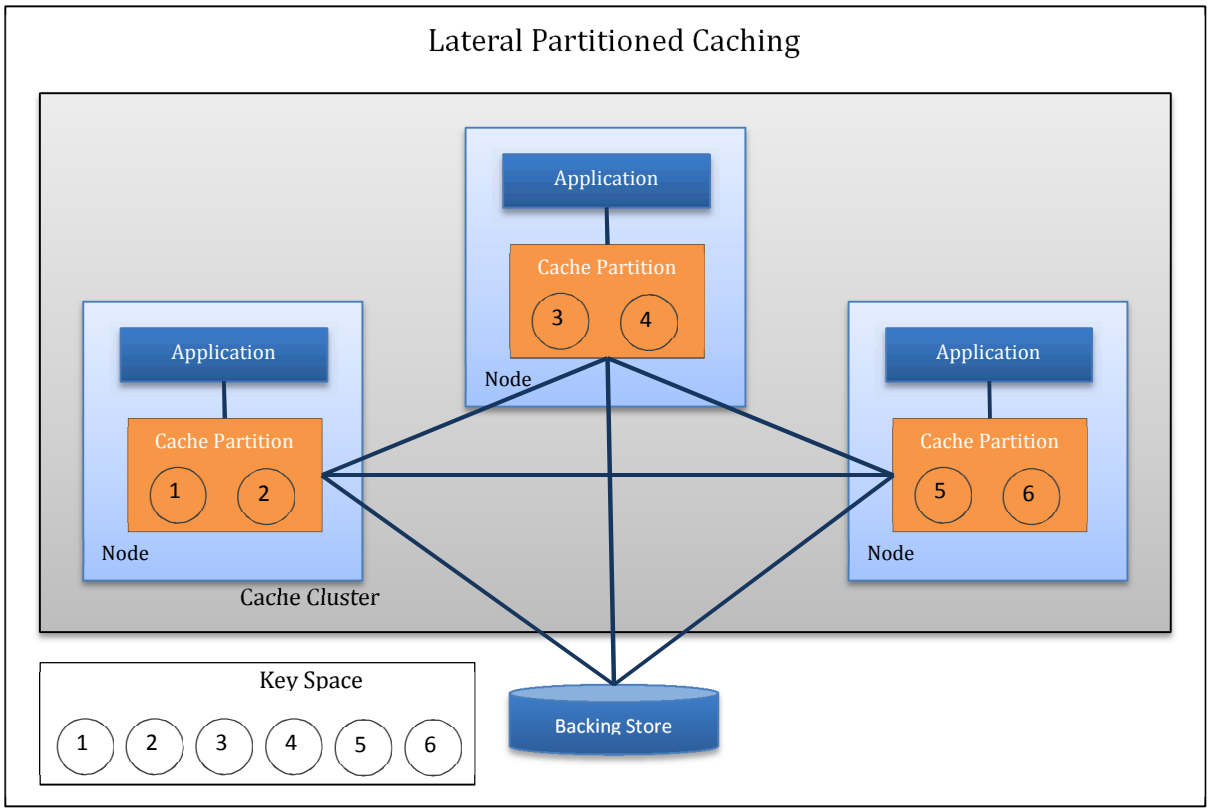

Figure 18. Scalable Distributed Caching with Lateral Partitioning

E.2. Relevant Standards

Currently, there are no established standards addressing NoSQL data stores. Distributed data caching standards are available for the Java platform. JSR 107 defines a caching standard and JSR 347 defines a data grid standard.

E.2.1. JSR 107

JSR 107 is the Java Community Process standard for temporary caching using the Java platform, also known as JCache. JCache specifies a distributed caching model with an API based on the Java ConcurrentHashMap. JCache also specifies integration with the Context & Dependency Injection (CDI) Java EE specification. [34]

E.2.2. JSR 347

JSR 347 specifies a Java Community Process standard for data grids using the Java platform. JSR 347 builds on the JCache specification, and addresses

additional concerns such as cache persistence, replication & distribution, eviction/expiration, transactions & concurrency control. [35]

E.3. Use Cases

Caching solutions are highly flexible and can be applied to a number of use cases. Examples include the following:

- Simple Non-Distributed Caching – stateful applications can improve access to state information by maintaining a simple in-process key-value cache. Caching solutions improve upon simple map data structures for this purpose by providing:
    - Configurable cache management policies, e.g. using expiration or eviction
    - Optional integration with data storage solutions (e.g. as an L2 cache for Hibernate ORM)
- Distributed Caching – Caching can be used as a form of inter-process communication to distribute data across processing components in a distributed architecture. Updates (insertions, modifications, deletions) are replicated across the distributed cache (including in-process near cache copies) either synchronously or asynchronously, based on requirements for cache consistency. Cached data can optionally be stored in an underlying storage solution with configurable consistency and availability.

  Subscription-based notifications can be used to alert processing components of cache updates of interest, providing an event-driven processing model. Many caching solutions provide cache event listeners and publish/subscribe messaging to support this capability.

This is the last page of the document.