SAND2014-20569R

US NDC Modernization

SAND-xxxx
Unclassified Unlimited Release
December 2014

US NDC Modernization Iteration E1 Prototyping Report: Processing Control Framework

Version 1.1

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

SAND-xxxx
December 2014

US NDC Modernization Iteration E1 Prototyping Report: Processing Control Framework

Ryan Prescott
Benjamin R. Hamlet

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

ABSTRACT

During the first iteration of the US NDC Modernization Elaboration phase (E1), the SNL US NDC modernization project team developed an initial survey of applicable COTS solutions, and established exploratory prototyping related to the *processing control framework* in support of system architecture definition. This report summarizes these activities and discusses planned follow-on work.

REVISIONS

Version	Date	Author/Team	Revision Description	Authorized by
1.0	3/21/2014	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 system architecture as a central 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 first iteration of the US NDC Modernization Elaboration phase (E1), spanning Q1 - Q2 FY2014, the prototyping effort included initial COTS surveys and exploratory prototyping addressing three core elements of the system architecture:

1. The *Common Object Interface (COI)* provides the system and research tools with access to persistent data via an abstraction of the underlying storage solutions.

2. The *processing control framework* provides for the definition, configuration, execution and control of processing components within the system, supporting both automated processing and interactive analysis.

3. The *User Interface Framework (UIF)* provides a flexible platform for the definition of extensible graphical user interface (GUI) components & composition of GUI displays supporting users of the system and research tools.

This report summarizes the iteration E1 prototyping activities of the SNL project team *specific to the processing control framework*. E1 prototyping activities for the COI and UIF are described in separate reports.

2. SCHEDULE

This report summarizes the processing control prototyping work completed during the three-month period from December 2013 to February 2014, based on the following schedule.

Period	Activity
December 2013	OSS/COTS survey
January – February 2014	Initial Exploratory Prototyping

3. MOTIVATION

Prototyping provides input critical in the definition of the system architecture, supporting selection of core software development languages and technologies, identification of architecture constraints & assumptions, and definition of high-level design patterns. In addition, the prototyping activity provides a foundation for development of the executable architecture deliverable.

4. PROCESSING CONTROL FRAMEWORK

4.1. Definition

The processing control framework is a software mechanism providing for the definition, configuration, execution and control of system processing components, supporting both automated and interactive analysis processing. The processing control framework includes the following elements:

1. An interface for defining automated processing components & processing topologies
2. A runtime environment supporting deployment, execution, monitoring and control of processing topologies

Note that the processing control framework may encompass multiple solutions supporting different types of processing within the automated and interactive analysis workflows (e.g. near-real-time vs. batch & interactive processing models).

4.2. Design Goals

- Provide a fault-tolerant, horizontally 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

4.3. Constraints

- COTS: Prefer Open Source Software (OSS) and other Commercial Off-The-Shelf (COTS) solutions to custom software development where available.
- Standards: Prefer solutions based on open standards wherever possible.

4.4. Iteration E1 Prototyping Activities

Iteration E1 prototyping activities focused on surveying COTS software solutions (principally open source software) addressing the requirements and constraints identified thus far for the processing control framework. Candidate solutions were identified through online research into available COTS/open source tools, and through discussions with other SNL project teams knowledgeable in COTS solutions for similar applications.

Note that the survey results presented here are not exhaustive; they represent an initial effort constrained to the available E1 schedule and staffing resources. Identification and evaluation of candidate software solutions is intended to be an ongoing activity during the elaboration phase, as development of the architecture definition and executable architecture prototype progress. Section 0 identifies additional survey work scheduled for iteration E2 and beyond.

The survey effort included a first-order assessment of candidates to eliminate those solutions not well suited to the US NDC and IDC applications. Additional investigation, including limited exploratory prototyping, was conducted for promising candidates not eliminated as part of the survey. Similar follow-on investigations may be conducted for other candidates as part of future work.

4.4.1. Initial COTS Survey

The candidates surveyed as part of the E1 prototyping work can be organized into four categories:

1. Stream Processor Frameworks

   Stream processing frameworks are largely focused on providing infrastructure for real-time data analytics on unbounded streams of continuously arriving data. They facilitate real-time, rather than batch mode, cluster based parallel computation. The stream processing frameworks surveyed are Apache Storm [1], Apache S4 [2], and Apache Samza [3].

2. Java Application Frameworks

   The Java EE [24] and Spring [25] application frameworks provide general support for the development, configuration, deployment and management of scalable, secure, distributed applications, including both automated and interactive processing. Both frameworks are widely used in industry, and are well suited for the development of server-side applications.

3. Enterprise Service Bus Frameworks

   Enterprise Service Bus implementations provide general support for systems built using Service Oriented Architecture (SOA). ESBs can be thought of as providing services used by the primary system services. The WS02 ESB [5] was surveyed for this prototype. Another ESB, Mule [6], was used in the SOA proof of concept project completed during Inception Iteration 2.

4. Complex Event Processors

   Complex Event Processors are similar to Stream Processors as they focus on real-time data analytics for continuously arriving data. Complex Event Processors are generally based around a query engine used to select stream data for processing. This approach lends itself to dynamic topologies that evolve with the processing results and data arriving on the stream whereas Stream Processors tend to use statically configured topologies that are run on all incoming stream data. The Esper Complex Event Processor [4] was surveyed.

The candidates surveyed reflect a move to modern development languages within the Java Virtual Machine (JVM) ecosystem, principally Java. The dominance of Java among the candidates is a reflection of its prominence within the solution space.

The candidates also reflect an alignment to the industry state of practice for mission critical application development where stability and maturity are important factors to be balanced against cutting edge innovation. Given the required longevity of the modernized system, COTS solutions with greater prevalence and larger development communities were preferred to newer, less well-established offerings.

Candidates were assessed based on the quality and applicability of their feature sets, as well as their maturity and the apparent strength of their user/development communities. Survey results are summarized for each candidate in the following sections.

4.4.1.1. Stream Processor Frameworks

The three Stream Processing Frameworks surveyed approach the problem of real-time data analytics for streams of incoming data in similar ways. Each of the products was actively developed by well-known internet companies, and each are currently open source projects managed by Apache. Because the products are so similar, differentiating among them involves down selection based on either specific implementations of a key feature or based on qualities like industry adoption, user support, or potential for longevity.

4.4.1.1.1. Apache Storm

Apache Storm is an open source project originally released to GitHub by Twitter in September 2011. It has been managed as an Apache Incubator project since September 2013. Storm is currently and actively under development. Storm 0.9.01, released in December, 2013, was used for this prototype.

Storm processing is organized as a topology (i.e. a data flow graph) of *spouts* that provide data and *bolts* that perform processing. Figure 1 depicts a simple Storm topology. While Storm runs in the JVM, it has several built in features supporting development in non-JVM languages. The processing topologies used by Storm are defined using a language independent format (Apache Thrift [12]) that allows Storm topologies to be developed in a wide range of programming languages. Storm also has built-in support for accessing *spouts* and *bolts* developed using non-JVM languages through the *multilang* protocol.

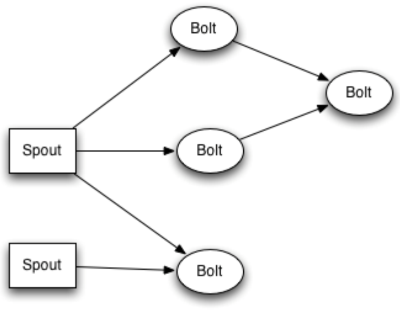


Figure 1. Storm topologies are directed graphs where the edges represent data *tuples* flowing between *spouts* and *bolts.* [1]

Storm has a pluggable messaging system with existing implementations for ZeroMQ [10] and Netty [11]. Even though these messaging systems do not have the strong reliability guarantees of projects like Kafka [7] and Zookeeper [9], Storm provides a variety of processing guarantees that include "all data is processed at least once" and "all data is processed exactly once". Processing components written by users shoulder some of the bookkeeping responsibilities to achieve "at least once" processing guarantees as Storm requires explicit acknowledgement when bolts finish processing a tuple. "Exactly once"

[1] Reproduced from the Storm tutorial (http://storm.incubator.apache.org/documentation/Tutorial.html), which is available on the Apache Software Foundation's Storm website [1].

processing guarantees are achieved through a separate topology definition API (known as Trident) that does not require bookkeeping in client codes.

Storm provides a custom cluster resource management system. This system uses Zookeeper to coordinate cluster nodes and is resilient to node and process failures. Resource allocation is defined statically but can be updated dynamically using a command line interface. Since this requires manual intervention, Storm does not provide automatic, dynamic, processing elasticity.

Storm has experienced high adoption in the open source community and is used by several prominent companies for data analytics tasks.

4.4.1.1.2. Apache S4

Apache S4 is an open source project released by Yahoo in October, 2010. It has been an Apache Incubator project since September 2011.

S4 topologies are defined in Java code as a graph of Processing Elements and Data Streams connecting the Processing Elements.

S4 uses Zookeeper for the communication layer, which provides persistent and durable messaging to S4. S4 processing topologies are written in Java. S4 implements a check-pointing system to provide processing guarantees and to prevent data loss.

S4 uses Apache Helix [13] for cluster resource management, load balancing, dynamic resource scalability, and fault tolerance.

Despite being available publicly for several years, S4 has not fostered widespread interest in the open source community. This fact brings long term support and development of the project into question.

4.4.1.1.3. Apache Samza

Apache Samza is an open source project released publically in September, 2013. It was originally developed by LinkedIn to support their real-time data analytics needs.

Samza topologies are defined in configuration files. There is no built-in support for running non-Java components within processing topologies. Samza can provide guarantees that “all data is processed at least once” or “all data is processed exactly once” by a topology without exposing the bookkeeping tasks or states to client codes.

Samza uses Apache Kafka for messaging. Kafka provides reliable messaging with guaranteed message delivery. Kafka is itself a distributed application and provides parallel message processing, load balancing, and certain message delivery order guarantees. Since Kafka messages are backed by Zookeeper, which writes to disk, Samza nodes or processes can fail without losing messages and without messages that are pending delivery overwhelming the available system memory. Keeping all messages persisted to disk facilitates data replay through Samza topologies.

Samza uses Apache YARN [8] for resource management. YARN is a compute cluster resource management and task scheduling utility originally developed for the popular Hadoop map-reduce batch processing project. Samza benefits from YARN's load balancing, node management, and process and resource isolation features. Samza does not support dynamic elasticity for cluster resource allocation.

Samza has not been available as open source software long enough to accurately gauge future community interest, support, or long-term viability.

4.4.1.2. Java Application Frameworks

The Java application frameworks surveyed in E1 represent the two most prevalent enterprise java application development solutions: Java Enterprise Edition (Java EE), and Spring.

Java EE and Spring were included in the survey because both provide technologies for the development of modular processing architectures composed of loosely-coupled, distributed processing components interacting through well-defined interfaces. Java EE and Spring provide a very similar set of capabilities, and generally follow similar design patterns. Spring has influenced the recent evolution of a number of Java EE standards.

It should be noted that both Java EE and Spring provide many additional capabilities of interest to the modernized system architecture that fall outside the scope of the processing control framework and so are not addressed here. One example relates to data persistence abstractions (e.g. Java Persistence API, Object Relational Mappings & Data Access Objects), which are addressed separately as part of the Common Object Interface (COI) prototyping effort. Another example relates to client presentation frameworks (e.g. Java Server Faces, Struts and Spring Model-View-Controller), which are addressed as part of the user interface framework prototyping effort.

Both Java EE and Spring are typically deployed using an application server, which provides an implementation of the core functions of the application framework. An assessment of the most prominent application servers is included in Section 4.4.1.2.3.

4.4.1.2.1. Java EE

Java EE is the standard enterprise Java computing platform. It provides a widely-supported, open standard enabling the development of vendor and platform-agnostic software. Java EE specifies a comprehensive set of technology standards addressing multi-tiered, scalable, reliable Java applications. Among the standards, the E1 survey focused on those related to enterprise application development, including:

- Enterprise Java Beans (EJB) provide an architecture for the development and deployment of component-based applications.
- Contexts & Dependency Injection (CDI)/Dependency Injection for Java provide a flexible application configuration model minimizing coupling between components.
- Interceptors provide support for managing Aspect Oriented Programming (AOP) functions such as logging, auditing, and profiling.
- Java Transaction API (JTA) provides support for transactions within the application.
- Java Messaging Service (JMS) supports messaging between processing components using reliable, asynchronous, loosely coupled communication

Java EE is highly stable, mature technology with large and well established user and development communities.

4.4.1.2.2. Spring Framework

The Spring framework is an open source application framework for the Java platform. Whereas Java EE provides a set of standards for which multiple implementations are available, the Spring framework provides a set of concrete technologies with many capabilities similar to those available in Java EE implementations. Unlike Java EE, the Spring framework is not based on open standards.

Among the Spring framework technologies, the E1 survey focused on those related to enterprise application development, including:

- Spring Inversion of Control (IoC) Containers provide an architecture for the development of component-based applications, as well as a flexible application configuration model similar to Java EE EJBs and CDI.

- Spring Aspect Oriented Programming (AOP)/AspectJ, which provide support for managing Aspect Oriented Programming (AOP) functions similar in capability to Java EE Interceptors.

Support for other relevant capabilities addressed by the Java EE standard (e.g. transaction management, messaging) are provided through integrations with third party solutions, including Java EE implementations of JTA and JMS.

The Spring framework provides an alternative to Java EE for the development of Java applications. As with Java EE implementations, the Spring framework is highly stable, mature technology with large and well established user & development communities.

4.4.1.2.3. Application Servers

Application servers provide a runtime environment for the execution of application software. In the case of Java, the application server acts as an extension of the JVM that provides core application services such as support for connection pooling, clustering, fail-over, and load-balancing.

A number of open source and commercial application servers are available which provide support for Java EE and/or Spring applications. In the case of Java EE, these applications provide implementations of the Java EE standards. In the case of Spring, they provide an enhanced runtime environment for applications built on the Spring framework.

It should be noted that both Java EE and Spring applications can be developed to execute outside of the application server. This capability is provided by default as part of the Spring framework. Java EE provides an embedded container accessible from the Java Standard Editions (SE) environment that implements a subset of Java EE standards, including a light-weight EJB implementation, as well as support for transactions, security and AOP concerns (interceptors).

The most prevalent application servers providing support for Java and/or Spring are described briefly in the sections below.

4.4.1.2.3.1. Wildfly

Wildfly [26, 27], formerly known as JBoss AS, is an application server developed by the JBoss division of Red Hat. Wildfly provides full support for the latest Java EE standards (Java EE 7), as well support for deployment of Spring applications. Wildfly is free and open source software. Red Hat provides optional commercial support for JBoss Enterprise middleware on a subscription basis. Wildfly is a mature, stable and widely adopted solution [29] that has been under active development since 1999.

4.4.1.2.3.2. GlassFish

GlassFish [14] is an application server originally developed by Sun Microsystems and currently managed by the Oracle Corporation. Glassfish is the reference implementation of Java EE, and provides complete support for the Java EE standards. It also provides support for deployment of Spring applications. GlassFish is open source software.. A commercial version known as Oracle GlassFish Server is also currently provided by Oracle.

In 2013, Oracle announced that it will discontinue commercial support for GlassFish. Although the community version will be supported at least through version 5 (the current version is 4) and will serve as the reference implementation through at least Java EE 8 (the current version is 7), the longer term future of GlassFish is uncertain.

4.4.1.2.3.3. Apache Tomcat

Apache Tomcat [15] is a web server developed by the Apache Software Foundation (ASF). Although it is often considered together with Java EE application servers owing to its support for the Java Servlet and JSP standards, it does not provide full support for the Java EE standards, notably excluding support for Enterprise Java Beans[2]. Tomcat does provide support for deployment of Spring applications. Tomcat enjoys a large user base and development community; it is the most widely used web application server on the market [29] and presents a compelling environment for Spring applications. Tomcat is free and open source software.

4.4.1.2.3.4. Jetty

Jetty [16] is a web server project developed as part of the Eclipse Foundation. As with Tomcat, Jetty it is often considered together with Java EE application servers owing to its support for the Java Servlet and JSP standards; however it does not provide full support for the Java EE standards, notably excluding support for Enterprise Java Beans. It does provide support for deployment of Spring applications. Jetty enjoys a large user base, and its popularity is on the rise [29]. Jetty is free and open source.

4.4.1.2.3.5. WebLogic

WebLogic Server [17] is a proprietary Java EE platform developed by the Oracle Corporation that provides a number of large-scale enterprise solutions, including a Java EE application server, web portal, Enterprise Application

[2] Apache provides a Java EE compliant application server known as TomEE, which combines Apache Tomcat with additional Java EE support, including OpenEJB, OpenJPA and others. TomEE is a recent offering (2012) and has yet to establish a significant user community beyond that of Tomcat. It was not evaluated as part of the E1 survey.

Integration (EAI) platform, transaction server (Tuxedo), telecommunication platform and web server. WebLogic Server is a robust, mature and comprehensive solution for large-scale enterprise applications. As with IBM WebSphere, it is considered to be a more complex and heavy weight solution than the other application servers surveyed [28].

4.4.1.2.3.6. WebSphere

WebSphere [18] is a proprietary suite of enterprise application integration middleware developed by IBM. At the center of the WebSphere product line is the WebSphere Application Server, which provides a robust, mature & comprehensive solution for large-scale enterprise applications. As with Oracle's WebLogic Server, the WebSphere application server is considered to be a more complex and heavy-weight solution than the other application servers surveyed [28].

4.4.1.2.3.7. Conclusions

Overall, Java EE and Spring provide very similar capabilities for the development of Java applications. Both are mature and stable technologies. Spring is more widely used than Java EE; however both enjoy a large user base and developer community [29].

Standardization presents an important distinction between the two. Java EE is a widely-supported, open standard enabling the development of vendor and platform-agnostic software. In contrast, Spring is a non-standard, open-source solution developed and maintained by a single commercial vendor.[3] Spring has significantly influenced the Java EE standard, and is considered by some to be a de facto standard.

All of the application servers considered in the E1 survey are mature, robust products. As shown in Figure 2, Tomcat is the most widely used of the servers assessed. However, Tomcat does not include support for a number of important Java EE standards, most notably EJBs. Tomcat is compelling as a Spring deployment technology. Among the application servers providing full Java EE support, Wildfly (formerly JBoss AS) is the most widely used, making it a compelling choice for Java EE applications. Although GlassFish is the current reference implementation of Java EE, Oracle's discontinuation of commercial support calls into question its long-term viability.

WebSphere and WebLogic are intended for large-scale enterprise application development, and thus are likely more complex and heavy weight solutions than required for the modernized system architecture.

[3] Spring is currently developed by Pivotal (www.gopivotal.com).

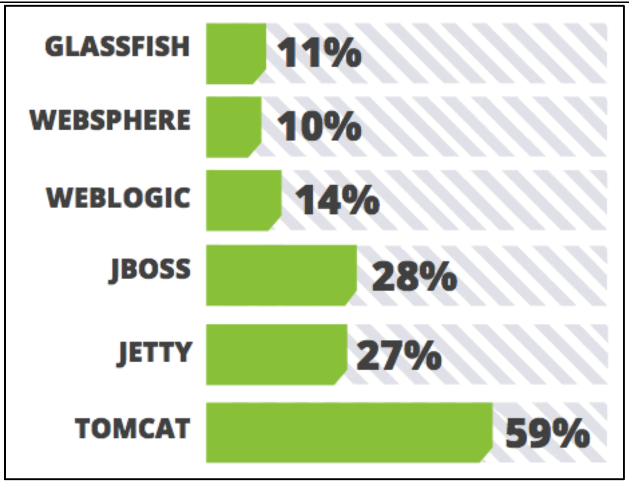


Figure 2. Java Application Server Usage Based on 2012 Developer Survey[4]

4.4.1.3. Enterprise Service Bus

ESBs are general purpose integration solutions supporting service oriented systems. ESBs are typically used in systems where there is a need to integrate a variety of products that were created in disparate programming languages, which run on different types of hardware, or which are provided by separate organizations. The common theme in this scenario is there may not be a common system architecture used when implementing the individual services and yet there is a need and benefit to using them together. The ESB's goal is to bridge the gap between such heterogeneity. ESBs are most commonly used with web services and were studied as part of a SOA proof of concept project completed during Inception Iteration 2. MuleESB was used in that project and therefore was not surveyed as part of exploratory prototyping. In general, MuleESB provides similar features to those studied in the WS02 survey.

4.4.1.3.1. WS02

WS02 is a Java ESB based on Apache Synapse. The first version of WS02 was release in 2007, though Apache Synapse has been available since 2005.

WS02 provides end point (i.e. service) failover and load balancing and can be integrated with a separate WS02 product, the WS02 Elastic Load Balancer, to achieve elastic load balancing at the level of either the ESB or of individual end points. Support for implementing data processing guarantees is available through transactional messaging which can be used to guarantee message delivery.

[4] Reproduced from the article "Developer Productivity Report 2012: Java Tools, Tech, Devs & Data" By Oliver White. See the References section for full citation.

WS02 facilitates processing topology definitions through a type of end point referred to as a mediation sequence. When a mediation sequence receives a message it will execute a sequence of other mediators in a user-defined order. Built in mediators are available for common tasks such as sending messages to services, aggregating messages received from services, providing conditional message routing to services, and providing priority-based service execution. Custom mediators can be written by users for other processing tasks.

Since ESBs provide a central hub facilitating service integration, distributed parallel processing, support for multiple programming languages, secure messaging, and interoperability with persistence mechanisms, all are possible and supported at various levels by WS02 and other full featured ESBs.

4.4.1.4. Complex Event Processor Frameworks

4.4.1.4.1. Esper

Esper is a real-time data processing system that works on streams of incoming data. Esper, like other complex event processors, differentiates itself from the stream processors (e.g. Storm, Samza, S4) in that the programming model is based on a specialized query language enabling selection of subsets of data from the incoming stream for processing rather than on processing all arriving data with a fixed topology.

Esper is open source and has been under development since 2004. It is managed by EsperTech and has seen adoption by a number of prominent companies (including PayPal and Raytheon). EsperTech provides an enterprise version of Esper that supports clustering. A separate project, EsperHA, provides high availability and guarantees no data loss.

Esper is built around a powerful, efficient, and expressive query engine. Queries executed against the engine can be likened to database queries occurring on a non-persistent, transient stream of incoming data. Data selected for processing can be injected back into the stream for exposure to other selection queries. Processing topologies in Esper would likely be crafted in client code as a series of data selection queries. Even though Esper's query engine is analogous to those provided by databases, Esper itself does not provide a data persistence mechanism.

4.4.2. Exploratory Prototyping

Based on the results of the initial survey, two of the candidates were selected for further investigation and exploratory prototyping in the E1 timeframe: Apache Storm and Java EE/Wildfly.

As discussed in Section 4.4.2.1, the feature set provided by the stream processor candidates holds promise for the processing control requirements of the system. Storm was selected as the most well-established and widely used of the stream processors. In contrast, Samza is a new offering that has yet to establish a significant user base and the S4 project appears to be largely dormant.

ESB candidates were not investigated further, given that that these technologies, and MuleESB in particular, were assessed as part of the SOA proof of concept project completed during Inception Iteration 2.

The event query paradigm central to the complex event processor solutions was judged to be too specialized for the more general processing needs of the system's automated and interactive analysis processing. Thus, these candidates were not investigated further in E1.

A Java EE implementation was selected for further investigation rather than the Spring Framework based on the stated preference for solutions built on open standards. As discussed in Section 4.5, further investigation of the Spring framework has been identified as potential follow-on work. Wildfly was selected for the Java EE exploratory prototype as the most widely used application server providing full Java EE support.

Table 2 in Appendix A provides a summary comparison of feature sets between the E1 prototype candidates and existing systems.

4.4.2.1. Apache Storm

4.4.2.1.1. Background

Apache Storm is a framework that manages real-time, distributed, fault-tolerant, and multi-language data processing applications. Storm is real-time as it is meant to process data as it becomes available rather than storing data for batch processing at a later time. Storm is distributed to support processing high data loads and to provide resilience to machine failures. Storm is fault tolerant as it can provide both "at least once" and "exactly once" processing guarantees for each piece of data it processes. Storm is multi-language since the fundamental definitions for processing components and processing topologies are language independent.

4.4.2.1.1.1. Processing Model

A Storm topology is a directed graph (either cyclic or acyclic) where each node is either a *spout* or a *bolt*. Storm uses *spout* nodes to emit data into topologies, so these nodes have zero incoming edges and one or more outgoing edges to other nodes in the topology. *Spout* nodes will typically interface with files or applications external to the Storm topology to access the data they emit. Storm uses *bolt* nodes to process data within a topology. Since *bolts* can emit their results into the topology for further processing, *bolts* have one or more incoming edges and zero or more outgoing edges to other nodes in the topology. Storm passes data *tuples* along the edges between nodes. *Tuples* can contain any number of data elements and there are no limitations on data types except that all data types must have a serialization so that Storm can pass data between processes and machines.

Storm exploits two types of parallel processing. First, multiple *spouts* and *bolts* in a topology can simultaneously process different *tuples*. Second, Storm can replicate *spouts* and *bolts* to multiple processing tasks. This allows simultaneous execution of the same underlying *spout* or *bolt* code on different data tuples.

4.4.2.1.1.2. Clustering

Storm topologies run on Storm clusters. Storm clusters contain a master node and one or more worker nodes. The master node runs Storm's *Nimbus* daemon which is responsible for distributing the *spout* and *bolt* code used within topologies to the worker nodes, assigns tasks to workers, and monitors the worker nodes for failures. Each worker node runs the *Supervisor* daemon which receives data arriving at the machine and assigns it to worker processes. Each worker process contains one or more *Executor* threads, and each *Executor* runs one or more *tasks*. A *task* is either a *spout* or a *bolt*. The *Supervisor* is also responsible for starting and stopping worker processes on the machine.

Storm has a custom cluster manager that determines which worker nodes perform which processing tasks. As discussed in section 4.4.2.1.3.3, Storm supports some user specified groupings that allow clients to indicate where a component should run relative to other components. Figure 3 shows a simple Storm topology with one *spout* and two *bolts* running on a cluster with four worker nodes. Each worker process has several *Executor* threads which run processing *tasks.* The *tasks* are color coded according to which *spout* or *bolt* they run. Storm assigns the work of each *spout* and *bolt* to one of the available tasks as data becomes available for processing. Storm's ability to distribute a topology's work in this manner is how it provides horizontal scalability.

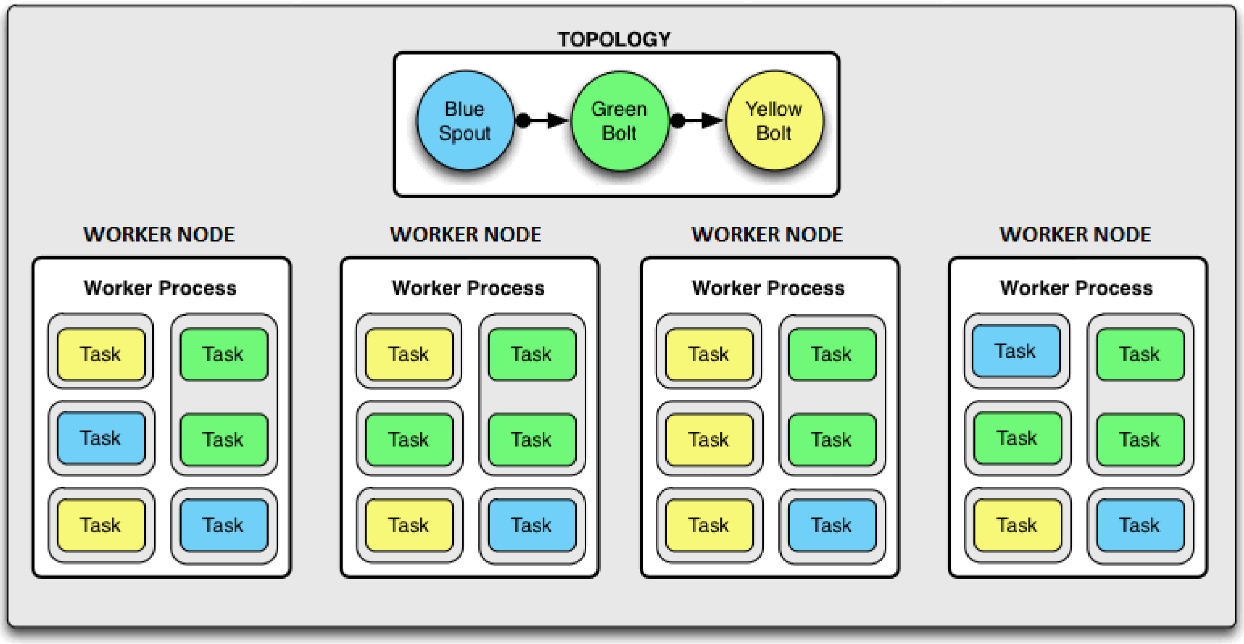


Figure 3. Mapping Storm processing topologies to clusters[5]

Storm *Nimbus* and *Supervisor* daemons use Zookeeper and the local file system on each cluster node to store processing states. Zookeeper is used to coordinate the runtime cluster configuration and state parameters required to determine how topologies are distributed across machines and how the current work is distributed across the worker nodes. Storm uses the local file system to store larger chunks of distributed data, such as topology definition files. Storing processing state outside of Storm allows the Storm daemons and Storm workers to independently fail and restart without causing failures to the running topologies. Zookeeper also runs on a cluster, providing a layer of data resilience and availability in the presence of machine failures. Storm does not expose these persistence mechanisms to clients, so client codes are responsible for handling their data management and persistence needs independent of Storm.

4.4.2.1.1.3. Fault Tolerance

Storm implements a reliability model that can guarantee all data emitted from a *spout* is processed by a topology. However, if Storm is used in a broader system where a *spout* is a consumer accessing data from an external producer, Storm cannot guarantee that all data created by the producer are consumed by the *spout* or that all data consumed by the *spout* are emitted into the topology. These kinds of processing guarantees are attainable if the Storm *spout* has durable access to external data. These types of processing guarantees are available in the exploratory prototype through use of a Storm *spout* that

[5] Adapted from Michael Noll's *Understanding the Parallelism of a Storm Topology* (http://www.michael-noll.com/blog/2012/10/16/understanding-the-parallelism-of-a-storm-topology/) [31].

consumes data from the Apache Kafka fault tolerant producer-consumer messaging framework.

4.4.2.1.1.4. Process Monitoring

Storm provides several process monitoring and diagnostic tools. Command line tools are used to start, stop, rebalance, and perform simple introspection on the Storm cluster and the topologies it is running. Storm also has a web client providing information on cluster resource availability, consumption, and uptimes; topology resource consumption and uptimes; and worker node uptimes. Additionally, all Storm daemons and worker processes write detailed log files, including status updates and exception stack traces, to a common directory.

4.4.2.1.2. Cluster Configurations

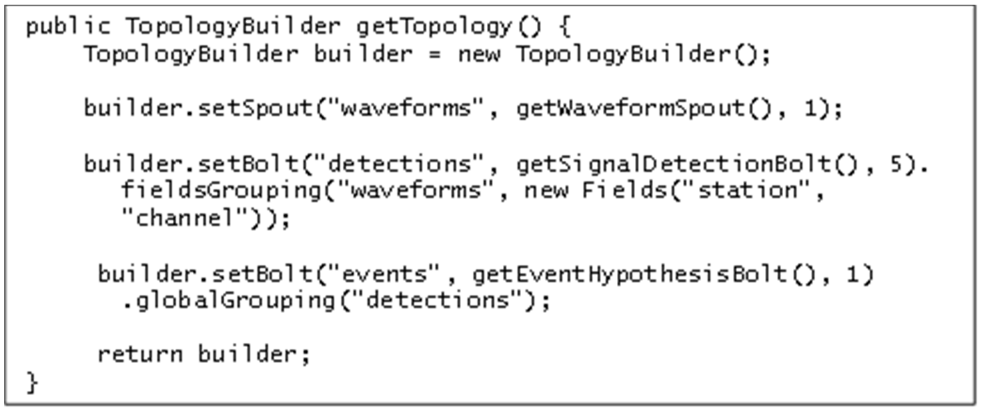
The Apache Storm exploratory prototyping effort involved configuring a two node Storm cluster capable of running a variety of Storm processing topologies. One node in the two node cluster was configured as both a master and a worker while the second node was exclusively a worker node. The master node therefore ran Storm *Nimbus*, Storm *Supervisor*, Zookeeper, and Kafka while the worker node only ran Storm *Supervisor*. An operational cluster would likely involve multiple nodes forming a Zookeeper cluster, multiple nodes forming a Kafka cluster (if Kafka was used), and additional *Supervisor* nodes. Storm *Nimbus* cannot be distributed, so it must run on a single node regardless of cluster size.

Adding a new worker node to a Storm cluster is a straightforward process that minimally involves installing Storm, installing Storm's messaging library, and editing a Storm configuration file with references to the Zookeeper and Nimbus machines. Since Storm is designed to immediately close the JVM, and therefore close the *Supervisor*, whenever an exception occurs it is also recommended to run Storm under process supervision. The prototype machines were configured to use the process supervisor named *supervisor*. Process supervision was configured to start the Storm *Supervisor* on the worker node and Storm *Nimbus*, Storm *Supervisor*, Kafka, and Zookeeper on the master node. Since Storm can restart worker tasks after failure and the process supervisor can restart Storm after it fails, running Storm under process supervision provides a durable processing cluster. A convenient side effect of process supervision is all processes required to run the master and worker nodes are started when the process supervisor is started, so only one step per machine is required to start the cluster.

4.4.2.1.3. Topology Configurations

4.4.2.1.3.1. Prototyped *spouts* and *bolts*

Each processing topology created for this prototype consisted of three tasks: one *spout* task producing waveform *tuples*, one *bolt* task consuming waveform *tuples* and producing signal detection *tuples*, and one *bolt* task consuming signal detection *tuples* and producing event hypothesis *tuples*. All of the data produced in these tasks was randomly generated. The tasks did not run actual algorithms to detect signals or build event hypotheses. Figure 4 lists the Java code used to define these topologies. The listing shows how each Storm component is assigned a textual identifier that is then used to link it to other components (e.g. the "detections" *bolt* received input from the "waveforms" *spout*. The numbers at the end of each `setSpout()` and `setBolt()` call indicate the number of tasks running the associated component.



```
public TopologyBuilder getTopology() {
    TopologyBuilder builder = new TopologyBuilder();

    builder.setSpout("waveforms", getWaveformSpout(), 1);

    builder.setBolt("detections", getSignalDetectionBolt(), 5).
      fieldsGrouping("waveforms", new Fields("station",
      "channel"));

     builder.setBolt("events", getEventHypothesisBolt(), 1)
      .globalGrouping("detections");

     return builder;
}
```

Figure 4. Sample Storm code to define a processing topology.

Two types of waveform *spouts* were created. The first is a standalone Java implementation that generates and emits random waveforms. The second is a Java implementation that consumes waveforms from Kafka and emits them into the topology. This *spout* works in conjunction with a Java application that generates random waveforms and publishes them to Kafka.

Two signal detection *bolts* were created that consume waveforms and produce signal detections. The first is a Java implementation that emits random signal detections on the consumed waveforms. The second is a C++ implementation performing the same function.

One event formation *bolt* was created that consumes signal detections and produces event hypotheses. This *bolt* has a Java implementation.

Four types of Storm topologies were defined using these *spouts* and *bolts*:

Table 1: Topology Definitions

	Waveform *spout*	Signal Detection *bolt*	Event Formation *bolt*
Topology A	Standalone	Java	Java
Topology B	Kafka	Java	Java
Topology C	Standalone	C++	Java
Topology D	Kafka	C++	Java

4.4.2.1.3.2. Specifying Processing Guarantees

Each of the topologies in Table 1 was defined using two different reliability models. The first group of topologies used the default "at most once" model which guarantees that all data emitted into a topology is processed no more than one time, but does not guarantee that all data emitted into a topology is processed. The second group of topologies used the "at least once" model which guarantees that all data emitted into a topology is processed at least one time, but does not guarantee that the data is only processed one time. Neither of these models guarantees any particular data processing order, so Storm may process a tuple emitted into a topology either before or after subsequent tuples emitted into the topology. When the Kafka waveform *spout* is used the system can additionally guarantee that all of the waveforms published to Kafka through the external program are consumed by the Kafka waveform *spout* and emitted to the topology, at which point the Storm reliability model dictates whether or not the waveform *tuples* are guaranteed to get processed by the topology.

Configuring a topology to use the "at least once" reliability model was a trivial change from the "at most once" model that only required tagging each tuple with an identifier as it was emitted to the topology and subsequently acknowledging each time a bolt completes processing for a tuple. Storm uses this information to keep track of the potentially expansive tuple graph stemming from each tuple emitted by a *spout* and will reemit a tuple when it determines processing has failed. Storm uses timeouts to recognize unresponsive nodes, workers, or tasks and automatically fails processing for their assigned tuples. Storm users can also explicitly fail tuples within their *bolt* implementations.

Storm can also provide the stronger guarantee of "exactly once" processing which guarantees each tuple emitted into a topology is processed by the topology a single time, but does not guarantee that tuples are successfully processed in the same order they were emitted. Storm topologies with "exactly once" processing semantics are defined using Storm's Trident API. These semantics also require a secondary database to store the state Storm uses to track tuple processing. Implementing Trident topologies is therefore significantly different than configuring normal Storm topologies and was not explored in this prototype.

Storm topology definitions allow specifying the number of worker processes running a topology, the initial number of executor threads for a particular *spout* or *bolt,* and the total number of tasks allocated for that *spout* or *bolt*. Since topology definitions are static these parameters place an upper bound on the cluster resources consumed by an executing topology. Multiple topologies can run simultaneously on the same cluster, but Storm does not provide a means for users to control provisioning specific worker nodes to specific topologies. Storm does provide a command line tool to dynamically change the parallelism of a running topology by setting the number of workers available to the topology and the number of executors assigned to each *spout* or *bolt*. Storm provides no built in means to automatically rebalance topologies, and therefore does not provide dynamic processing elasticity.

4.4.2.1.3.3. Assigning *tuples* to Processing Tasks

Though Storm does not allow specifying which nodes run which Storm components, it does allow clients to specify how the data processed by each component is distributed across the tasks running that component. Clients assign a textual name to each *spout* and *bolt* in the topology definition. These names are used to specify the fundamental data flow (e.g. the waveform *spout* provides input the signal detection *bolt* which provides input to the event hypothesis *bolt*). The names are also used to inform Storm of how the topology expects data to be routed across the individual tasks running each component (e.g. group data output from the waveform *spout* such that waveform data from a station is always processed by the same signal detection *bolt* task). A client can specify various types of groupings, including that all tuples with the same value for a particular field be routed to the same task, that tuples be distributed randomly but evenly across tasks, that tuples be routed to all tasks, and so on.

4.4.2.1.4. Type System

Storm has a weak type system. Storm components declare by name (not by type) which fields it will emit, but does not specify which fields it expects as input. This allows Storm components to be wired together in arbitrary ways and allows components to accept different types of tuples as input. Client codes that create topologies are responsible for connecting components in sensible ways.

Storm's Java APIs provide runtime type checking through the tuple accessor methods.

4.4.2.1.5. Multilanguage Support

Storm topologies are implemented as Apache Thrift structures. Since Thrift has bindings for most common programming languages (including Java, C++, C#, and Python) it is possible to write Storm topologies using a variety of programming languages. Since Storm runs on the Java Virtual Machine and provides a Java API for building topologies, it is straightforward to write Storm code in Java and other JVM languages. In practice, defining topologies in non-JVM languages will likely involve using a topology building API written in that language to provide an abstraction over the Thrift syntax. For example, the Petrel open source project can be used to create Storm topologies in Python.

Storm supports *spouts* and *bolts* written in non-JVM languages through the *multilang* protocol. Storm uses Java wrappers to communicate JSON messages over stdin and stdout with processes written in non-JVM languages. Storm defines the communication protocol and messaging format, which must also be implemented in the non-JVM language. The Java wrappers define the *spouts* and *bolts* for configuring in topologies and implement the Java half of the multilang protocol. The C++signal detection *bolt* written for the exploratory prototype uses the *multilang* protocol by leveraging the open source StormCPP implementation. One disadvantage for Unix or Linux based system is executable files distributed by *Nimbus* to cluster nodes lose their executable status. The effect of this is that *spouts* and *bolts* compiled as executable programs either require workarounds to be made executable or must be distributed to the cluster nodes independent of Storm.

A second approach to accessing C or C++ codes from Storm topologies is to use *spouts* or *bolts* accessing external executable code through the Java Native Interface. This approach circumvents Storm's *multilang* protocol by placing messaging responsibility on the implementation. However, it does not require any more Java *spout* or *bolt* implementations than the multilang protocol and has the advantages of avoiding JSON serialization and the potential to use local memory when passing parameters between the Java and native codes.

4.4.2.1.6. Serialization and Messaging

Storm uses Kryo to serialize and deserialize tuples to and from messages, but defaults to using the less efficient Java serialization if a Kryo serialization cannot be performed for an object. Client codes can define specific Kryo serializers to use on a class by class basis. Storm's messaging framework is pluggable with existing implementations in ZeroMQ and Netty. Storm does not guarantee reliable messaging between nodes. However, as discussed previously, Storm's reliability model can guarantee that all data emitted into the topology is processed, even if individual messages fail.

4.4.2.1.7. Summary

Storm provides a cluster management and distributed processing control mechanism. Topology definitions and resource allocations are statically configured, so all data flows and parallelism are configured before running the topology. Dynamic but manual intervention is required to update a running topology's resource allocation. Storm natively supports processing components built in a variety of languages, but JVM languages seem easier to use and it is not clear the JSON *multilang* protocol provides much advantage over JNI when accessing native codes. Storm can enforce a variety of processing guarantees. As a stream processor, it seems most suited to continuously running processing tasks that have little or no critical path reliance on external processing.

Storm has generated interest in the open source community and has over 8,000 GitHub stars. While Storm provides some documentation, much of the support required to setup, configure, and run the prototype cluster and topology came from online user groups and third party tutorials. Storm support may evolve now that it is an Apache managed product.

4.4.2.2. Java EE/Wildfly 8

4.4.2.2.1. Background

4.4.2.2.1.1. Containers & EJBs

Java EE provides a modular component framework for creating robust, distributed applications. At the heart of this framework is the concept of a *container*, which provides foundational services that otherwise would need to be implemented within the application software (e.g. transaction handling, state management, multi-threading, resource pooling). The container also manages the lifecycle of the contained application component and supports dependency injection. Java EE specifies multiple container types based on its multi-tier architecture, including Application Client Containers, Web Containers, and Enterprise Java Bean (EJB) Containers.

Java EE specifies two types of EJBs that can execute within the EJB container:

1. *Session Beans* encapsulate processing logic that is invoked from client views (local, remote, web services) as part of a client session. Three types of session beans are defined.

    - *Stateful Session Beans* interact with a single client and maintain state information across client calls for the duration of a session.

- *Stateless Session Beans* can interact with multiple clients, but do not maintain conversational state across client invocations.

- *Singleton Session Beans* are instantiated once per application, exist for the lifecycle of the application, and provide concurrent, shared access across multiple clients.

2. *Message Driven Beans* (MDBs) support asynchronous processing of messages, acting as JMS listeners. MDBs encapsulate logic that is executed as part of a message-driven architecture.

The E1 exploratory prototype focused on an automated processing architecture composed of Message Driven Beans executing within a set of EJB containers.

4.4.2.2.1.2. Dependency Injection

Dependency Injection is a central feature of Java EE enabling flexible configuration of the application and minimal coupling between components. Through dependency injection, the dependencies between components are assigned dynamically at runtime, minimizing the need for explicit static coupling, and allowing for alternative dependencies based on the deployment context of the application (e.g. allowing mock versions of dependent components within test environments). Dependency injection can be specified using either XML configuration or (more commonly) through the use of Java annotations within the source code. The E1 exploratory prototype incorporated dependency injection annotations to specify mock seismic processing component implementations and JMS message topics for communication among components.

4.4.2.2.1.3. Wildfly Server Configuration

The Wildfly server can be configured to run on one of two modes: standalone and domain. In standalone mode, each instance of the Wildfly server is configured, deployed and managed independently. In domain mode, a set of Wildfly servers is configured, deployed and managed as a group, known as a managed domain. Although both modes support HA clustering, managed domains simplify clustered server management by maintaining a consistent configuration and coordinating updates across the cluster. As shown in Figure 5, within a domain, the server cluster is managed by a designated Domain Controller, which coordinates the other servers by way of a Host Controller instance running on each host.

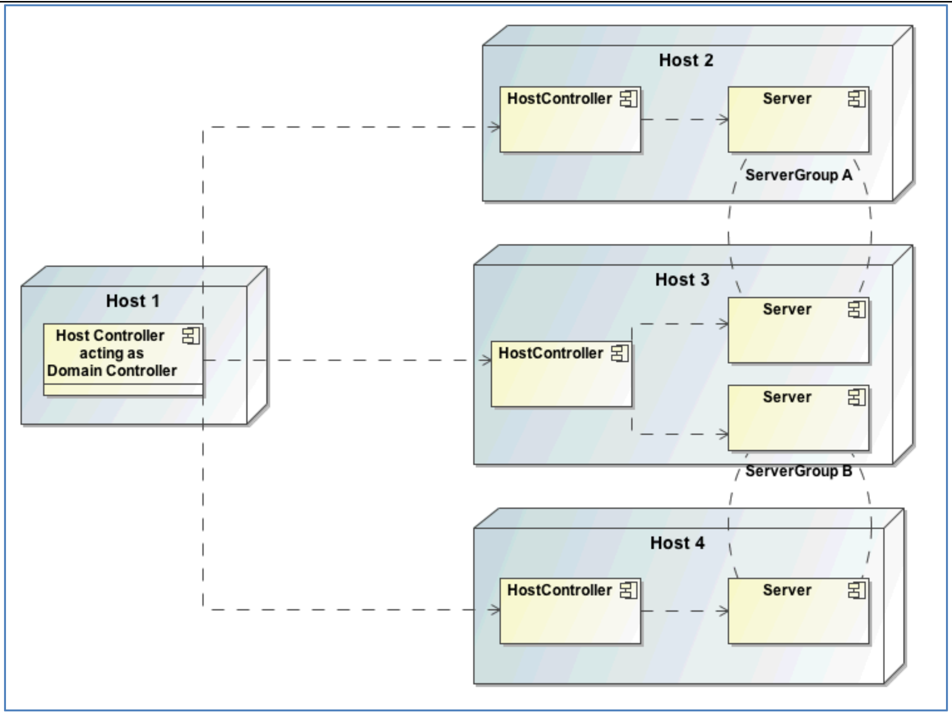


Figure 5. Wildfly Clustering Using Managed Domains[6]

Configuration of the managed domain is defined using an XML file: domain.xml. Among other things, this file contains the cluster definition and configuration settings defined as part of a HA profile.

Wildfly provides HA services through two features:

1. *Failover* – clients interacting with a server instance will not be interrupted, even if the node on which the instance is executing fails. Wildfly supports failover by providing distributed storage of the state information needed to restore processes, and by relocating processing across the cluster upon failure of the underlying process or node.

2. *Load Balancing* – client requests are distributed across the available nodes of the cluster to maintain timely response in the presence of high request volume.

4.4.2.2.1.4. Messaging

Java EE includes the JMS standard to provide for message-based communication between processing components within a distributed application. JMS provides multiple messaging patterns, including point-to-point messaging via message queues, as well as publish-subscribe messaging via message topics. Although

[6] Reproduced from the "Wildfly Admin Guide" By Heiko Braun. See the References section for full citation.

Wildfly supports integration with a number of message oriented middleware technologies, HornetQ is provided as the default solution. The E1 prototype used HornetQ publish/subscribe topics for communication among components.

HornetQ includes a number of High Availability (HA) facilities to provide *at least once* or *once and only once* message delivery guarantees. In order to support *at least once* delivery guarantees, HornetQ includes shared persistence of message data across nodes within an HA cluster. Two alternate approaches are provided:

- *Message Replication* – Messages are replicated on multiple nodes within the cluster and are synchronized continuously across the network. Upon failover of the designated *live* server, the designated *backup* server retrieves from its locally replicated messages in order to resume message processing.

- *Shared Store* – Messages are persisted in a shared storage area accessible from the nodes of the HA cluster - typically, a storage area network (SAN). Upon failover of the designated *live* server, the designated *backup* server retrieves messages from the shared storage area in order to resume message processing.

In order to support *once and only once* delivery guarantees, HornetQ supports transactional processing, where any in-progress transactions on the failed node are rolled back and restarted using persisted messages on the backup node. HornetQ also provides a duplicate message detection capability to prevent repeated message processing during failover for non-transactional deployments.

4.4.2.2.2. Exploratory Prototype

For the E1 exploratory prototype, an HA cluster was defined across 2 Virtual Machine (VM) hosts using a managed domain. Each VM was configured to host an instance of the Wildfly Server running a mock seismic processing pipeline. The mock seismic pipeline was defined as a set of Message Driven Beans communicating mock waveform and signal detection data via a set of HornetQ publish/subscribe JMS topics. The purpose of this prototype was to further investigate the primary features of a processing architecture built using EJB, CDI and JMS within a Wildfly application server cluster. Findings based on the prototype are discussed in the sections below.

4.4.2.2.2.1. Wildfly Server Management

Definition & deployment of the Wildfly Managed Domain HA cluster was relatively straightforward given the solid documentation and quick start examples available from the JBoss community. Wildfly provides a sophisticated Command Line Interface (CLI) for administration of the runtime environment, however, for the E1 prototype, only a small set of basic commands were needed to deploy and manage the application. A JBoss Eclipse plugin is available, which supports execution of the Wildfly server and deployment of the Java EE application via Maven from within the Eclipse IDE. Wildfly also provides a secure, web-based administration interface supporting configuration and monitoring.

4.4.2.2.2.2. Mock Seismic Pipeline

The mock seismic pipeline was defined as a collection of processing components encapsulated within Message Driven Beans. The components themselves were implemented as Plain Old Java Objects (POJOs), which were injected into their respective MDBs at runtime using CDI. Depending on their role within the pipeline, components consumed mock waveform and/or signal detection data, and produced either mock waveform data, signal detections, and/or event hypotheses. The mock inputs and outputs were transmitted between processing components asynchronously using JMS topics injected into the MDBs at runtime using CDI. For convenience, a servlet was defined to inject mock waveform data into the front end of the pipeline by way of a client web page. Figure 6 depicts the mock pipeline processing components, data provider servlet, and JMS message paths.

Although it was not implemented as part of the E1 pipeline, the prototype was designed in such a way that the processing components could be instantiated separately within Session Beans as part of a mock interactive analysis interface.

Development of the MDBs, JMS interfaces and processing components was straightforward and benefitted significantly from a wealth of solid documentation and quick-start example code available from the JBoss community.

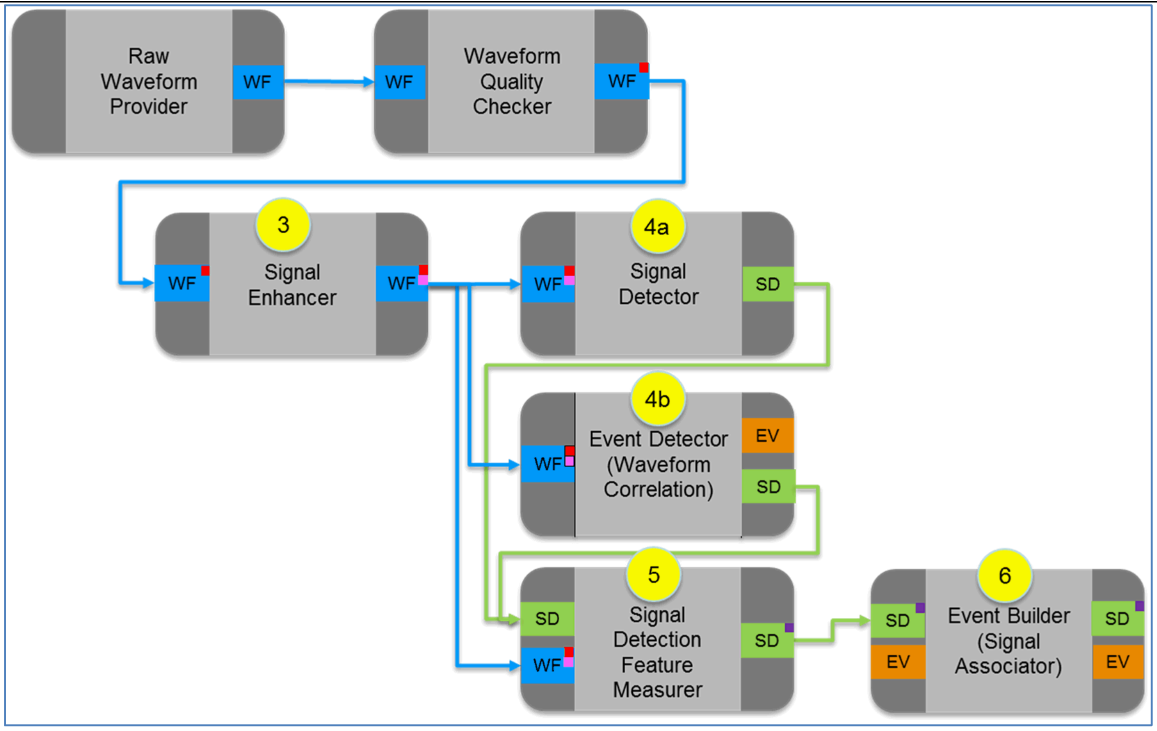

Raw Waveform Provider
WF
Waveform Quality Checker
WF
3
Signal Enhancer
4a
Signal Detector
SD
4b
Event Detector (Waveform Correlation)
EV
5
Signal Detection Feature Measurer
6
Event Builder (Signal Associator)

Figure 6. E1 Prototype Mock Seismic Pipeline

4.4.2.2.2.3. Limitations

A significant limitation of Java EE that was discovered as part of the evaluation is the lack of support for execution of native software within Java EE applications. The EJB standard prohibits loading of native libraries (e.g. via Java Native Interfaces or Java Native Access) from within the EJB container. Similarly, the standard prohibits the explicit execution of threads within the EJB-managed applications. These restrictions in the standard are intended to preserve the security, stability and portability of Java EE applications; however, they eliminate the ability of the application to directly invoke processing components implemented in non-JVM languages such as C and C++.

One option for overcoming this limitation is to use web services to communicate between the Java EE applications and non-JVM components of the system. This solution was not included as part of the E1 prototype and should be addressed in follow-on work in order to assess the viability of a Java EE-based architecture for mixed-language applications pending a decision regarding the scale and usage pattern of non-JVM software within the modernized system architecture.

4.4.2.2.2.4. Conclusions

An architecture built from the Java EE technologies used in the E1 prototype provides a flexible, robust, fault-tolerant distributed processing capability that is well suited to development of a Java-based automated processing pipeline.

Harnessing additional Java EE technologies such as Session Beans will likely provide similarly powerful capabilities for the development of interactive analysis processing components.

The Wildfly server provides a stable, secure and user friendly runtime environment enabling deployment and management of highly-available Java EE processing applications.

Together the Java EE APIs and Wildfly server administration tools, documentation and examples enable highly efficient application development.

A significant limitation of the Java EE architecture is the lack of support for execution of non-JVM software components. Solutions for this use case should be evaluated in follow-on work in order to understand the viability of Java EE technology in the modernized system architecture.

4.5. Follow-On Work

Four areas of follow-on work have been identified for iteration E2 and beyond based on the E1 prototyping activities:

1. Explore additional PCF solutions, based on input from the community
2. Assess custom PCF solutions
3. Develop a basic processing pipeline prototype
4. Select a PCF solution for the executable architecture prototype

4.5.1. Explore additional PCF solutions

In iteration E2, the prototyping team will solicit and incorporate candidate solutions from the community. The intent is to leverage design and prototyping knowledge available from applicable development organizations and to ensure that informed decisions are made in the development of a processing control framework considering a breadth of candidate solutions.

Emphasis will be placed on existing design patterns and associated solutions that provide flexible, multi-language processing support with minimal coupling between components.

4.5.2. Assess custom PCF solutions

In iteration E2, the prototyping team will assess custom solutions meeting the requirements of the processing control framework. The purpose of this effort is to determine whether a solution incorporating more substantial custom software will better meet the needs of the system than the COTS-based solutions explored in iteration E1.

4.5.3. Select a PCF solution for the executable architecture prototype

Based on the E1 and E2 exploratory prototyping, the team will select a solution for use in developing the PCF element of the executable architecture prototype. This selection will be completed by the end of the E2 iteration, in order to allocate sufficient time for iterative development of the executable architecture prototype in iterations E3-E4.

4.5.4. Develop a Basic Processing Pipeline Prototype

Beginning in E2, the prototyping team will assemble a representative set of seismic processing components based on software baselines drawn from GNEM-related research projects and possibly from existing US NDC and IDC applications. These components will provide more representative processing software for use in further evaluation of candidate PCF solutions, performance benchmarking, and development of the executable architecture prototype.

APPENDIX A. COMPARISON OF PROTOTYPE AND EXISTING SYSTEM PROCESSING CONTROL FRAMEWORKS

Table 2: Feature sets of E1 Prototype Candidates and Current Systems

Feature	E1 Prototype Candidates		Existing Systems			
	Apache Storm	**Java EE/Wildfly**	**US NDC (Current)**	**IDC (Current)**	**NEIC (Current)**	**SeisComp3**
Processing Deployment & Execution	Apache Storm / Apache Zookeeper	Wildfly	Custom	Custom	Custom	Custom
Processing Definition	Apache Storm, Apache Thrift	Wildfly	Custom	Custom	Custom	Custom
HA Clustering	Apache Storm / Apache Zookeeper	Wildfly mod_cluster	N/A	N/A	N/A	N/A
Transaction Management	Trident	Wildfly JTA	N/A	N/A	N/A	N/A
Messaging	Multiple: ZeroMQ, Netty, Kafka	Multiple JMS providers (default: HornetQ JMS)	Oracle Advanced Queuing	Tuxedo	Custom	Spread

Table 3: E1 Survey Results Summary

Category	Candidate Solution	Summary Assessment
Enterprise Java Application Frameworks	Java EE	Advantages: Widely-used open standards with large development community. Provides a robust platform for development of scalable, fault-tolerant, distributed processing architectures. Disadvantages: EJB standard prohibits use of native libraries and direct thread creation, limiting design options supporting non-JVM languages.
	Spring Framework	Advantages: Widely-used open-source solution with large development community. Provides a robust platform for development of scalable, fault-tolerant, distributed processing architectures. Disadvantages: Not standards-based.
Stream Processors	Apache Storm	Advantages: Open-source solution with significant industry interest. Provides a robust platform for development of scalable, fault-tolerant, distributed processing architectures. Supports multiple development languages. Disadvantages: New offering. Not standards-based.
	Apache Samza	Advantages: Provides a robust platform for development of scalable, fault-tolerant, distributed processing architectures. Disadvantages: New offering that has yet to establish significant industry interest. Not standards-based. Does not support multiple languages(Java only).
	Apache S4	Advantages: Provides a robust platform for development of scalable, fault-tolerant, distributed processing architectures. Supports multiple development languages. Disadvantages: Little industry interest and development activity. Not standards-based.
Enterprise Service Bus	WS02 ESB	Advantages: Provides a robust platform for integration of heterogeneous systems via standardized messaging as part of a service-oriented architecture. Disadvantages: Design strengths not well aligned to the end-state modernized architecture (US NDC and IDC are not heterogeneous system of systems).
Complex Event Processor	Esper	Advantages: Provides a robust platform for development of scalable, fault-tolerant, distributed processing architectures. Disadvantages: Specialized, query-based architecture does not fit system processing needs particularly well. Not standards-based. Does not support multiple languages (Java only).

REFERENCES

1. *Storm: Distributed and Fault Tolerant Realtime Computation*, The Apache Software Foundation, (http://storm.incubator.apache.org/).

2. *S4: Distributed Stream Computing Platform*, The Apache Software Foundation, (http://incubator.apache.org/s4/).

3. *Samza*, The Apache Software Foundation, (http://samza.incubator.apache.org/).

4. *Esper,* EsperTech Inc., (http://www.espertech.com/products/esper.php).

5. *WS02 Enterprise Service Bus*, WS02, (http://wso2.com/products/enterprise-service-bus/).

6. *Mule ESB,* MuleSoft Inc., (http://www.mulesoft.org).

7. *Kafka: A High-Throughput Distributed Messaging System*, The Apache Software Foundation, (http://kafka.apache.org/).

8. *Apache Hadoop NextGen MapReduce (YARN)*, The Apache Software Foundation (http://hadoop.apache.org/docs/r2.3.0/hadoop-yarn/hadoop-yarn-site/YARN.html).

9. *Zookeeper*, The Apache Software Foundation, (http://zookeeper.apache.org/).

10. *ZeroMQ,* iMatix Corporation, (http://zeromq.org/).

11. *Netty*, The Netty Project, (http://netty.io/).

12. *Apache Thrift*, The Apache Software Foundation, (http://thrift.apache.org/).

13. *Apache Helix*, The Apache Software Foundation, (http://helix.apache.org/).

14. *Oracle Glassfish*, Oracle Corporation, (http://www.oracle.com/us/products/middleware/cloud-app-foundation/glassfish-server/).

15. *Apache Tomcat*, The Apache Software Foundation, (http://tomcat.apache.org/).

16. *Jetty*, The Eclipse Foundation, (http://www.eclipse.org/jetty/).

17. *WebLogic*, Oracle Corporation, (*http:*www.oracle.com/WebLogic).

18. *WebSphere*, IBM, (www.ibm.com/software/websphere/).

19. *Petrel*, (https://github.com/AirSage/Petrel)

20. *C++ Wrapper for* Storm, 2012, (http://demeter.inf.ed.ac.uk/cross/stormcpp.html).

21. *Storm-Kafka*, (https://github.com/nathanmarz/storm-contrib).

22. Noll, Michael, *Running a Multi-Node Storm* Cluster, 2013 (http://www.michael-noll.com/tutorials/running-multi-node-storm-cluster/).

23. Hamlet, Benjamin R., et. al., *US NDC Modernization: Service Oriented Architecture Proof of Concept*, Sandia National Laboratories, Albuquerque, NM 87185, December 2014.

24. Jendrock, Eric, Cervera-Navarro, Ricardo, Evans, Ian, Haase, Kim, Markito, William, Srivathsa, Chinmayee. "Java EE 7 Tutorial." *Oracle Java EE Documentation*. Oracle, September 2013. Web. http://docs.oracle.com/javaee/7/tutorial/doc/home.htm.

25. Wheeler, Willie & White, Joshua, *Spring in Practice*, Manning Publications, 2013. Print.

26. Braun, Heiko. "Wildfly Admin Guide." *Wildfly 8 Documentation*. JBoss, Jan 22, 2014. Web. https://docs.jboss.org/author/display/WFLY8/Admin+Guide

27. Braun, Heiko. "Developer Guide." *Wildfly 8 Documentation*. JBoss, Jan 22, 2014. Web. https://docs.jboss.org/author/display/WFLY8/Developer+Guide

28. Maple, Simon, Shelajev, Oleg, Muuga, Sigmar, White, Oliver. "The Great Java Application Server Debate with Tomcat, JBoss, GlassFish, Jetty & Liberty Profile." *RebelLabs*. RebelLabs, May 21, 2013. Web. http://zeroturnaround.com/rebellabs/the-great-java-application-server-debate-with-tomcat-jboss-glassfish-jetty-and-liberty-profile/

29. White, Oliver. "Developer Productivity Report 2012: Java Tools, Tech, Devs & Data." *RebelLabs*. RebelLabs, May 15, 2012. Web. http://zeroturnaround.com/rebellabs/developer-productivity-report-2012-java-tools-tech-devs-and-data/

30. O'Grady, Stephen. "New Relic and the State of the Stacks." *RedMonk*. Redmonk, June 13, 2012. Web. http://redmonk.com/sogrady/2012/06/13/new-relic-stack-data/

31. Noll, Michael. *Understanding the Parallelism of a Storm Topology*, 2012 (http://www.michael-noll.com/blog/2012/10/16/understanding-the-parallelism-of-a-storm-topology/).

This is the last page of the document.