SAND2014-20567R

US NDC Modernization

SAND200X-XXXX
Unlimited Release
December 2014

US NDC Modernization: Service Oriented Architecture Proof of Concept

Version 1.1

Benjamin R. Hamlet, Andre V. Encarnacao, Keilan R. Jackson, Ian A. Hays, Nathan E. Barron, Luke B. Simon, James M. Harris, and Christopher J. Young

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

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

SAND200X-XXXX
Unlimited Release
December 2014

US NDC Modernization: Service Oriented Architecture Proof of Concept

Benjamin R. Hamlet, Andre V. Encarnacao, Keilan R. Jackson, Ian A. Hays, Nathan E. Barron,
Luke B. Simon, James M. Harris, and Christopher J. Young
Next Generation Monitoring Systems
Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185-MS0401

Abstract

This report is a progress update on the US NDC Modernization Service Oriented Architecture (SOA) study describing results from a proof of concept project completed from May through September 2013. Goals for this proof of concept are 1) gain experience configuring, using, and running an Enterprise Service Bus (ESB), 2) understand the implications of wrapping existing software in standardized interfaces for use as web services, and 3) gather performance metrics for a notional seismic event monitoring pipeline implemented using services with various data access and communication patterns. The proof of concept is a follow on to a previous SOA performance study. Work was performed by four undergraduate summer student interns under the guidance of Sandia staff.

REVISIONS

Version	Date	Author/Team	Revision Description	Authorized by
1.0	10/31/2013	US NDC Modernization Team	Initial Release	M. Harris
1.1	12/19/2014	IDC Reengineering Team	IDC Release	M. Harris

CONTENTS

FIGURES

TABLES

NOMENCLATURE

DOE	Department of Energy
ESB	Enterprise Service Bus
IDC	International Data Center
IMS	International Monitoring System
REST	Representational State Transfer
SNL	Sandia National Laboratories
SOA	Service Oriented Architecture
US NDC	United States National Data Center
XML	Extensible Markup Language

1. INTRODUCTION

This report describes a Service Oriented Architecture (SOA) proof of concept project performed as a follow on to a previous SOA study [1]. This proof of concept focused on deploying previously developed seismic monitoring research software as web services, configuring the services into a basic seismic monitoring pipeline, and gathering performance metrics for several pipeline structures and communication formats.

1.1. Service Implementations

Seismic event monitoring pipelines used in this proof of concept are composed of several distinct services. Each service implements interfaces with well-defined contracts for the types of data consumed as input and produced as output. The final outputs of these pipelines are a list of seismic events with their associated signal detections. Rather than using a single, monolithic application to perform all processing, this proof of concept links together services performing particular aspects of event formation. Since the output of one service is the input to the next service, combining the services creates a pipeline where raw waveform data is provided as input and event lists are produced as output.

The existing Sandia developed research applications wrapped as services were:

- **WavePro (*Wave*form *Pro*cessor)** – a signal processing application designed to detect signals of interest in data recorded by both 3-component seismic stations and seismic array stations [2]. WavePro includes functions for waveform filtering, rotation, beaming, and signal detection.
- **PEDAL (*Probabilistic Event Detector, Associator, and Locator*)** – an event detection application that operates on collections of signal detections from a network of seismic stations [3].
- **LocOO3D (*Locator, Object Oriented, 3-dimensional* base model support)** – a seismic event location refinement application that operates on an existing event with a set of associated signal detections. LocOO3D supports a variety of Earth models, including SALSA3D (Sandia and Los Alamos 3-dimensional model) [4, 5].

Additional services were created for:

- **Pipeline configuration --** a web interface used to:
    - configure the type of pipeline to run and the data to run through the pipeline
    - initiate runs and cancel runs
    - view log files from previous runs.
- **Waveform injection** – used to play raw waveform data previously recorded by International Monitoring System (IMS) seismic array stations into the WavePro service.
- **Bulletin output --** used to write processing results into a relational database with the Center for Seismic Study (CSS) 3.0 schema [6].
- **Pipeline results review** – used to graphically and textually review pipeline results.. An interactive map displays monitoring station and event locations. Textual displays list event and signal detection results computed by the pipeline.

1.2. Service Communication and Interface Patterns

Services developed for this proof of concept implement two service communication patterns and two service interface standards described in a previous SOA study [1]. Performance measurements gathered for running the same series of seismic data through the four possible pipeline configurations are discussed in Section 3.

The two communication patterns were:

1. Centralized control logic: a central controller component brokers communication between services. All services receive input from the controller and pass results back to the controller. Services are completely decoupled from one another.

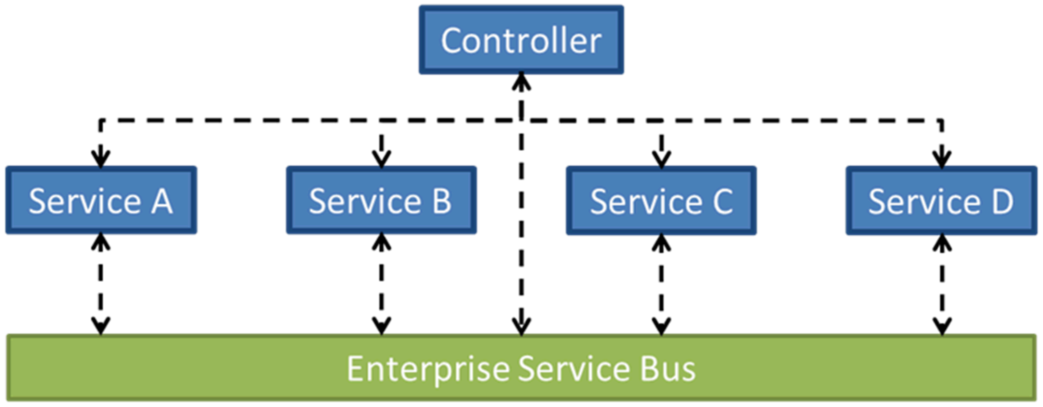

Figure 1. Centralized controller component

2. Distributed control logic: services pass messages directly to other services without using a central controller component.

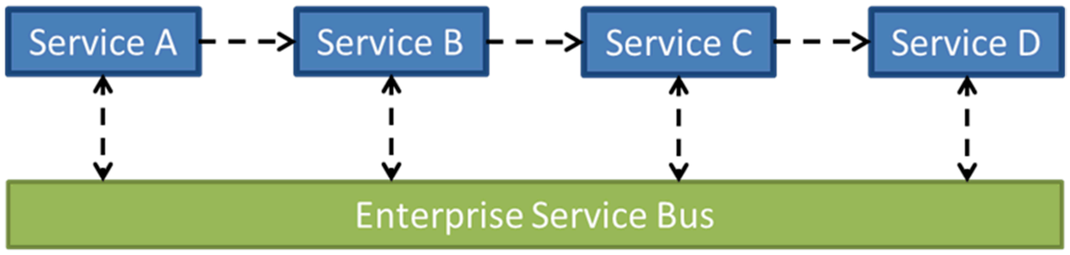

Figure 2. Distributed control logic (no centralized controller component)

The two service interface standards were:

1. *Light interfaces* follow the current system design, where information is passed primarily through the database. In this option, simple messages are sent between services declaring where in the database the invoked service will find input parameters. Service interfaces are decoupled from one another by passing parameters through the backing data store, but services rely on a contract external to the interfaces defining what parameters are expected in the data store.

2. *Rich interfaces* provide more service separation from the surrounding environment by defining all input parameters as part of the interface, freeing services from having to negotiate a common parameter area in the data store. Services optionally access the data store for additional configuration parameters that are either invariant from one invocation to the next or are implementation specific parameters hidden from clients.

1.3. Supporting Software

Additional software supported this proof of concept project. Of particular importance are a central clock to enforce timing constraints and a logging mechanism used to gather performance metrics and collect general status messages.

1.3.1. Central Clock and Available Data Lists

Each service in the proof of concept pipeline is configured to run at a fixed time interval. The timer intervals for each service are potentially unique. This creates a situation where services require two types of interfaces. The first accepts data for future processing and the second triggers processing on the available inputs. When run, a service processes all available inputs, creates outputs, and then passes those outputs to the next service for further processing. Each pipeline service registers a time interval with the central clock. The central clock notifies the service at the registered interval, triggering the service to process all previously queued data.

The central clock was implemented with a time scaling factor to allow the clock to run at speeds faster or slower than real time. This proved useful as it allows the pipeline be sped up to process, for example, one day's worth of data over the course of several hours rather than the day it would take if the clock were running in real time.

1.3.2. Performance Metrics

The pipeline processing services were configured to gather performance metrics during their execution. All times are computed by a single service running on a single machine so that valid elapsed times can be computed without requiring clocks to be synchronized across multiple machines. Both service execution and service overhead times are computed. Overhead time consists of the time required for messages to be prepared (marshaled from the service's internal data format into a standard format that can be communicated across a network), passed from one service to the next, and then parsed (unmarshaled from the communication format to the receiving service's internal format) at the receiving service. Execution time is the amount of time required to process a set of data and does not include any messaging, marshaling, or unmarshaling time.

As mentioned above, unless clocks are synchronized, elapsed times must be computed from initiation and completion times measured on a single machine. Since measuring overhead times includes the time required to send a message from one service to another, and since the services might be run on different machines, services are required to send an acknowledgement message whenever they receive a message whose overhead is being timed. The elapsed time for the message is then computed as the time required to marshal, transmit, and unmarshal the message plus the time required to send the acknowledgement message. Sending these acknowledgement messages introduces timing errors, but since these timing messages are small and are required for computing the overhead times for each service and each pipeline structure, the errors should be negligible when making relative performance comparisons between times measured for different pipeline structures.

1.3.3. Service Setup and Principles

Services in this proof of concept were implemented as standard web services using Representational State Transfer (REST) over the Hypertext Transfer Protocol (HTTP). Data passed between services were formatted as plaintext Extensible Markup Language (XML) documents. Seismic data were passed using an XML representation of CSS 3.0 formatted data.

The Mule Enterprise Service Bus (ESB) was used to configure and deploy services [7]. MuleESB is a widely-used free package that is easy to install and has good documentation. . MuleESB service flows were configured for the pipeline services and the central clock, central controller, and logging services used as utilities by the pipeline services. Additional flows were created to handle the logistics of configuring, starting, and stopping pipeline execution.

MuleESB provides a graphical editor for configuring services. Graphical versions of service flows are backed by XML descriptions that can also be hand edited. The PEDAL service seen in Figure 3 has a simple flow with two main steps and an additional asynchronous step for logging status messages and performance measurements. The first step, represented by the “HTTP” block, specifies where the service is located. It is used so that MuleESB and other services know where to access the PEDAL service. The second step, represented by the “REST” block, specifies the Java class used to implement the PEDAL service. It is used so that MuleESB knows what code to call when the PEDAL service is accessed.

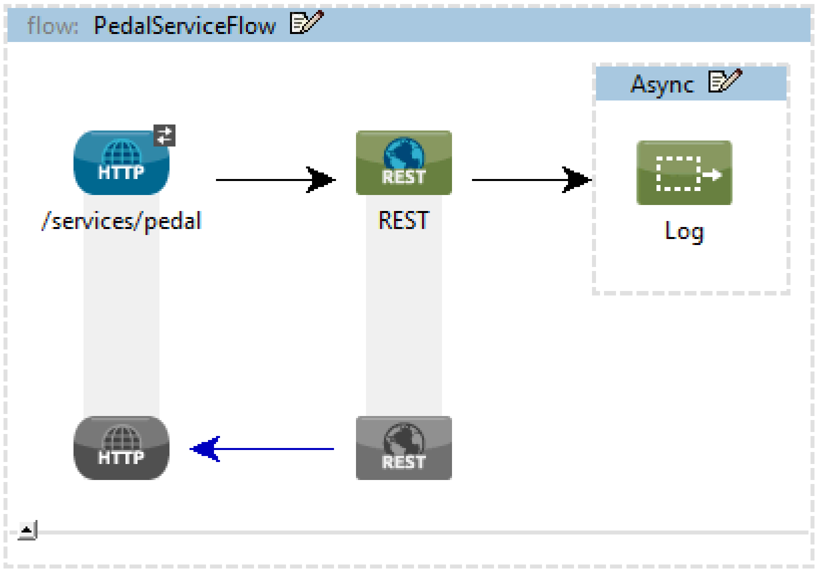

Figure 3. MuleESB flow describing the PEDAL service

The Java code used to define the PEDAL service implements a Java interface describing the contract required of any network event processing service used in this proof of concept system, which itself specializes a Java interface describing the contract required of all pipeline processing services. The basic service interface is:

```
public interface ServiceInterface<RichDataType,LightDataType> {
   public ESBMessage setProperties(ESBProperties properties);
   public ESBMessage richQueueData(RichDataType data);
   public ESBMessage lightQueueData(LightDataType data);
   public ESBMessage richPulse();
   public ESBMessage lightPulse();
   public ESBMessage isActive();
   public ESBMessage clearQueue();
}
```

Figure 4. Interface required by all pipeline processing services

This interface has a method for configuring the service (*setProperties*), methods supporting the central clock mechanism previously described with both the rich and light service interface standards (*richQueueData, lightQueueData, richPulse, lightPulse*), and methods to support either stopping a running pipeline (*clearQueue*) or discover when a pipeline is processing data (*isActive*)

The network event processing interface is a simple extension of the basic service interface that describes the type of data processed by any network event processing service:

```
public interface NetworkProcessingServiceInterface
         extends ServiceInterface<DetectionData,IdList> {
}
```

Figure 5. Interface required by all network event pipeline processing services

This interface forces network event pipeline processing services to process *DetectionData* objects as input to the rich interface (these objects contain signal detections at individual seismic stations) and *IdList* objects as input to the light interface (these objects describe a list of identifier used as primary keys in database tables and are used to query a database for signal detections).

Though simple, these interfaces help illustrate two basic principles followed throughout this proof of concept project:

1. *Decouple interfaces from implementations*: services are abstracted by generalized interfaces describing what types of data are processed by services. The interfaces are independent of underlying algorithm implementations. Since only a few simple methods are required to cover different realizations of a service interface, the data types used by the service methods must be sufficiently expressive to meet the needs of future implementations. This proof of concept used Java representations of CSS 3.0 objects, a standard format used in nuclear explosion monitoring seismology.

2. *Decouple services from one another*: services have no knowledge of the larger context in which they are being used. Services have no prior knowledge of which other services will call them nor do services have prior knowledge of where their outputs are sent. Each service's *setProperties* method is used at runtime to configure where the service will send its outputs. Service interfaces used in a production system would likely force stronger contracts between services by specifying the output data formats in addition to the input data formats, which would allow early indication when a service is configured to send its output to an incompatible location. This proof of concept project did not enforce this type of contract, so these kinds of errors were not discovered until one service attempted to send data of an incorrect type to another service.

Whereas both rich and light service interfaces are explicit in the Java code, there is no indication at the Java level of whether a centralized or distributed control structure is used. Pipeline control is configured at runtime by configuring each pipeline service to either send its output to another pipeline processing service or to send its output to the central controller service.

2. PROCESSING PIPELINE STRUCTURES

The seismic processing pipeline used in this proof of concept consists of five components executed in sequence:

1. Data injection: replays raw waveform data into the pipeline
2. Signal processing: signal detection processing on all received waveforms (WavePro)
3. Network processing: network event formation (PEDAL) processing on all signal detections.
4. Event relocation: refines network event locations (LocOO3D) produced by network processing.
5. a. Event bulletin: stores results to a database
   b. Event review: graphically presents results (uses the same interface as the event bulletin)

Each of these services implements the *ServiceInterface* previously described. The services are configured to communicate using either a central controller service or by direct communication, and data is passed either through a database (light interfaces) or directly between services (rich interfaces). This leads to four basic pipeline structures shown in Figure 6 and Figure 7.

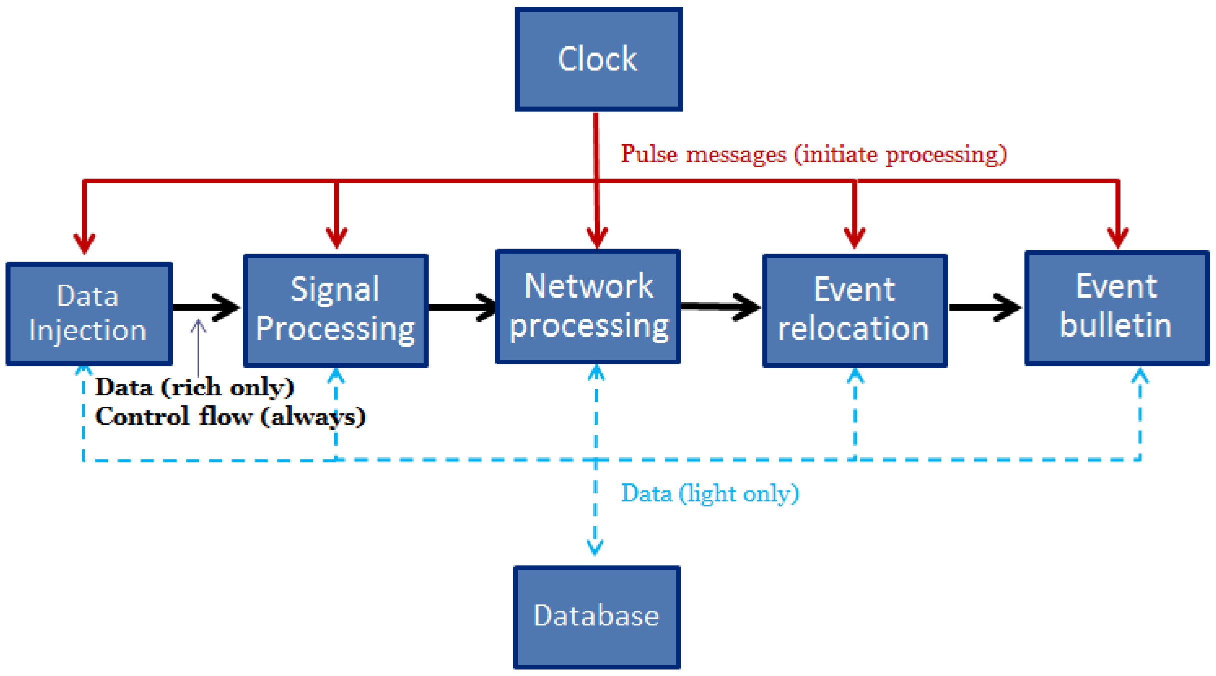

Figure 6. Processing pipeline with distributed control

When using distributed control, the pipeline services are configured to pass data to other pipeline services. When using light interfaces, the data is written to a database and a compact message containing only a list of database primary keys is passed to the next service. When using rich interfaces, the services pass data directly to the next service without going through a database. In either case, the receiving service queues data until it receives a message from the Clock indicating the service should process the available data.

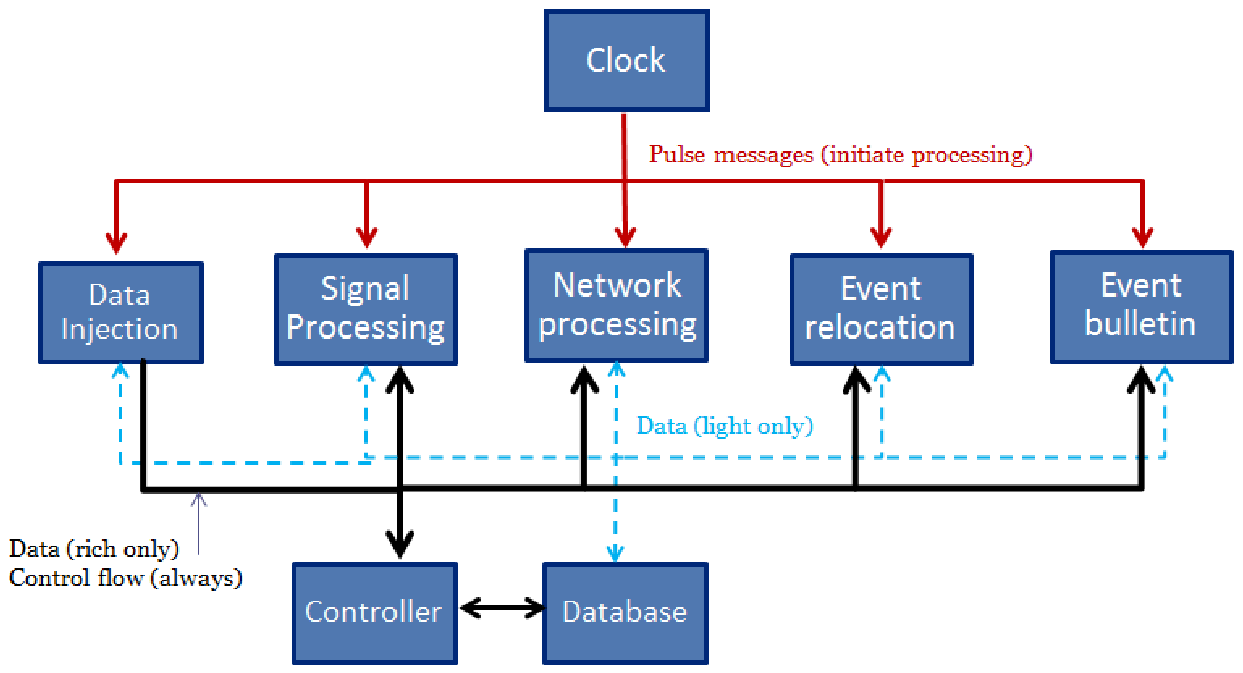

Figure 7. Processing pipeline with central controller

When using a central controller, the pipeline services are configured to pass data to the central controller service. This service has the option to parse the data and then make decisions about what type of processing to perform next. In this proof of concept the central controller simply passes the data to the next service in the pipeline. Although this means there is no logic in the controller, this setup is sufficient to study the performance implications of using a centralized control mechanism. Rich interfaces, light interfaces, and the clock service otherwise operate the same as when using distributed pipeline control.

Though not explicitly shown in either Figure 6 or Figure 7, all of the services are implemented and executed using MuleESB flows similar to the one shown in Figure 3.

When running these pipelines, each service is triggered by the clock service to process available data at fixed time intervals. Depending on the amount of raw data played into the pipeline by the data injection service, each service is likely to run multiple times.

3. PERFORMANCE STUDY AND RESULTS

As discussed above, the two types of service interfaces (rich and light) and two types of pipeline control (central and distributed) yield four possible pipeline configurations. A performance study comparing processing times to overhead times was performed to understand the implications of selecting one pipeline configuration over the other.

3.1 Configuration

Performance metrics were gathered for pipelines processing the 6 hours and 15 minutes of data recorded between April 20, 2006 at 23:23:20 GMT and April 21, 2006 at 05:38:20 GMT at 9

International Monitoring System (IMS) seismic array stations (SONM, MJAR, ILAR, YKA, BVAR, ASAR, WRA, GERES, and PDAR). Each service was registered with the clock to process data at fixed time intervals. The time intervals are listed in Table 1. The clock scaling factor was set to a speedup of 10 so that 60 minutes of data were played into the pipeline every 6 minutes. Performance metrics for message size, service processing time, and service overhead time were gathered and are discussed below. Services were run within MuleESB on a single workstation (Intel Xeon 3.60Ghz processor with 4 cores / 8 threads with hyperthreading enabled, 32.0GB of RAM, Windows 7).

Table 1. Service processing time intervals

Service	Processing time interval (s)
Data injection	10
Signal processing	300
Network processing	600
Event relocation	600
Event bulletin / event review	600

3.2 Results and Analysis

In the absence of synchronized clocks, the same clock must be used to measure the starting and ending times of a process to accurately measure an elapsed time. Service processing times can be measured by the service itself, but messaging times must be measured by the machine sending the message. Once values are measured they are assigned to services as follows:

- When using distributed pipeline control, messaging times and sizes are configured to measure the costs of invoking a service. In this case, message size and overhead time are both assigned to the receiving service.

- When using central pipeline control, messaging times and sizes are configured to measure the costs of invoking a service from the controller plus the cost of the subsequent call to the central controller after the service runs. In this case, the message size assigned to a receiving service is the combined size of the message received by the service plus the size of the subsequent processing results message sent to the central controller after the service runs. The message time assigned to a receiving service is the combined time to call the service from the central controller plus the time to send the results to the central controller after the service runs.

Full results are listed in Appendix A: Performance Testing Results. The overall results indicate waveform based operations dominate costs. In particular, the highest messaging times, messaging sizes, and processing times are all at the injector and signal processing services. While Table 1 indicates these services operate more often than the other services, the results in Appendix A also show they have average messaging times, messaging sizes, and processing times of the same order or higher than the non-waveform services.

When interpreting these results keep the following in mind:

- In a real monitoring system, it is likely that waveform data would be stored in small blocks to facilitate processing, but for SNL's nuclear monitoring research program we store waveform data in two hour blocks, to facilitate data management. The two hour blocks of data cause performance problems because less than one minute of data per channel is injected into the pipeline at a time. Reading two hours of waveform data, extracting the correct portion to inject, injecting that portion, and then unloading the rest of the waveform significantly degraded performance in early pipeline testing. This was mitigated by implementing a waveform cache to hold full two hour blocks of waveform data. Once data from a two hour block is read prior to the first injection of data, a waveform will likely appear in the cache and so subsequent injections will not incur either a database query to identify the correct waveform file or a disk read operation to load the waveform. Caching has implications for the processing times of services using the cache. Waveform caching is used by the injection service in pipelines using rich service interfaces and in the signal processing service for pipelines using light service interfaces.
- All services, including the ESB, were run on a single computer. Enabling hyperthreading allowed eight threads to run simultaneously on four cores. This scenario increases the possibility of resource contention among the pipeline services and between the pipeline services and other processes running on the computer. If resource contention occurs a service maybe blocked access to the machine's compute or input/output resources, increasing either messaging or processing times. Resource contention is not believed to have had significant effects on the performance metrics as the number of hyperthreads exceeds the number of services, the pipeline's time based execution spreads service execution out over time, and the computer's other loads were consistent during runs for each pipeline configuration
- Though not shown in this study's performance results, services could be easily deployed to different machines, taking advantage of a much more powerful distributed computing infrastructure. However, doing so has implications to messaging times and overall overhead costs.

Given these caveats, we make some observations based on the performance results:

- Total processing times for rich service interfaces are on the order of 5 times longer than total processing times for light services interfaces.
- Distributed pipeline control approximately doubles messaging overhead.
- Messages related to raw data availability are the most important factors in overhead rates
  - o Rich service interface performance is limited by waveform messaging times.
  - o .Light service interface performance is limited by data availability messages sent to the signal processor service.
  - o Injector messages account for significant overhead in each pipeline structure.

Rich service interfaces allow for isolated services that do not rely on access to an underlying data store. While this provides flexibility in system design it comes with increased messaging overhead. Since the costs associated with moving from light service interfaces to rich service interfaces are much higher than the costs associated with moving between controller styles for a given type of interface, the primary performance decision is selecting between rich and light service interface. The above observations indicate that if performance is of primary concern then light service interfaces should be preferred.

Our results show light service interfaces with central control took less overall time to run than light interfaces with distributed control, even though the overhead times were higher when using central control. Running more tests might better indicate the relative costs of distributed and central control when using light interfaces. Given the small message sizes associated with using light interfaces, the costs of merely passing a message to a central controller and then to another service do not fully capture the costs of a central controller with embedded logic. This is because higher control costs would occur if the central controller required access to data stored in a database before deciding how to react to a message. This is an area warranting further study if the performance penalty of using such a central controller is of concern.

4. SUMMARY

This proof of concept project successfully demonstrates the feasibility of implementing a seismic monitoring pipeline using service oriented architecture implemented using a standard ESB package (MuleESB). Generalized service interfaces based on standard seismic data representations were used to define basic pipeline operations that were then implemented using existing research applications. Abstractions for different types of system implementations (rich and light service interfaces, central and distributed pipeline control) allow various pipeline configurations. Each service gathered timing and messaging metrics to allow studying tradeoffs inherent in selecting one type of system architecture over another. Performance metrics gathered from running different pipeline configurations on previously recorded IMS waveform data provides information that can be used when making system architecture decisions.

5. REFERENCES

1. Hamlet, Benjamin R., A. V. Encarnacao, J. M. Harris, and C. J. Young, (December 2014). *US NDC Modernization: Service Oriented Architecture Study Status*, Sandia National Laboratories, Albuquerque, NM.

2. Encarnacao, Andre V. (2013). *WavePro*, Sandia National Laboratories internal discussions, May-September, 2013.

3. Draelos, Timothy D., S. Ballard, C. J. Young, R. A. Brogan, (2012). *Refinement and Testing of the Probabilistic Event Detection Association and Location Algorithm*, Proceedings of the 2012 Monitoring Research Review, Albuquerque, New Mexico, September 18-20, 2012.

4. LocOO Ballard, Sanford, J. R. Hipp, and C. J. Young (2009). *Efficient and accurate calculation of ray theory seismic travel time through variable resolution 3D earth models*, Seismic Research Letters 80, 6: 989–998, doi:10.1785/gssrl.80.6.989.

5. Ballard, Sanford. (2002). *Seismic event location using Levengerg-Marquardt least squares inversion*. SAND2002-3083, Sandia National Laboratories, Albuquerque, NM, (Unclassified)

6. Anderson, J., W. E. Farrell, K. Garcia, J. Given, and H. Swanger (1990). *Center for Seismic Studies Version 3.0 Database Schema Reference Manual*, SAIC Tech. Rep. C90-01, Arlington, Virginia.

7. Mule ESB. MuleSoft Inc., 2013 (http://www.mulesoft.org).

APPENDIX A: PERFORMANCE TESTING RESULTS

Table 2. Performance testing results

	RICH/DISTRIBUTED	RICH/CENTRAL	LIGHT/DISTRIBUTED	LIGHT/CENTRAL
Injector				
Messaging Time (ms)				
Message count	2,154.0	3,722.0	2,123.0	3,967.0
Total	449,931.0	2,088,254.0	283,528.0	913,235.0
Average	208.9	561.1	133.6	230.2
St. Dev	124.9	1,382.9	58.9	132.4
Lowest	102.0	106.0	95.0	92.0
Highest	2,213.0	81,031.0	1,160.0	1,999.0
Message Size (bytes)				
Message count	2,155.0	3,741.0	2,160.0	3,979.0
Total	252,135.0	3,358,300,265.0	252,720.0	31,905,135.0
Average	117.0	897,701.2	117.0	8,018.4
St. Dev	0.0	1,537,412.2	0.0	8,685.4
Lowest	117.0	117.0	117.0	117.0
Highest	117.0	2,308,379.0	117.0	41,981.0
Processing Time (ms)				
Message count	1,716.0	1,629.0	1,795.0	1,797.0
Total	48,133,028.0	58,919,501.0	1,670,258.0	1,699,716.0
Average	28,049.6	36,169.1	930.5	945.9
St. Dev	31,392.9	47,354.0	120.0	125.1
Lowest	4,193.0	3,643.0	851.0	849.0
Highest	154,540.0	188,783.0	3,954.0	4,065.0
Signal Processing				
Messaging Time (ms)				
Message count	1,743.0	1,744.0	1,831.0	1,904.0
Total	697,307.0	765,537.0	242,570.0	296,053.0
Average	400.1	439.0	132.5	155.5
St. Dev	139.2	143.1	49.1	72.6
Lowest	142.0	183.0	92.0	111.0
Highest	2,494.0	1,945.0	1,165.0	1,320.0

	RICH/DISTRIBUTED	RICH/CENTRAL	LIGHT/DISTRIBUTED	LIGHT/CENTRAL
Message Size (bytes)				
Message count	1,737.0	1,743.0	1,832.0	1,904.0
Total	3,649,942,446.0	3,438,507,061.0	30,897,854.0	108,043,424.0
Average	2,101,291.0	1,972,752.2	16,865.6	56,745.5
St. Dev	2,491,167.3	2,491,306.7	2,108.2	13,150.1
Lowest	46,167.0	46,167.0	13,779.0	116.0
Highest	2,309,256.0	2,308,379.0	29,879.0	141,809.0
Processing Time (ms)				
Message count	76.0	73.0	72.0	72.0
Total	1,081,348.0	1,024,285.0	7,697,305.0	6,845,740.0
Average	14,228.3	14,031.3	106,907.0	95,079.7
St. Dev	10,328.8	10,257.8	66,432.9	67,225.8
Lowest	3,236.0	170.0	11,819.0	11,710.0
Highest	54,285.0	47,319.0	238,187.0	255,862.0
Network Processing				
Messaging Time (ms)				
Message count	75.0	103.0	73.0	99.0
Total	19,132.0	40,249.0	14,346.0	26,691.0
Average	255.1	390.8	196.5	269.6
St. Dev	68.1	172.4	23.2	118.4
Lowest	145.0	182.0	117.0	142.0
Highest	569.0	883.0	278.0	1,004.0
Message Size (bytes)				
Message count	75.0	105.0	73.0	98.0
Total	81,581,887.0	79,055,677.0	17,821.0	20,072.0
Average	1,087,758.5	752,911.2	244.1	204.8
St. Dev	307,751.1	529,824.1	78.4	81.8
Lowest	415,115.0	2,704.0	116.0	116.0
Highest	2,164,232.0	1,757,545.0	506.0	476.0
Processing Time (ms)				
Message count	32.0	31.0	27.0	28.0
Total	180,027.0	132,824.0	94,541.0	72,706.0
Average	5,625.8	4,284.6	3,501.5	2,596.6
St. Dev	6,781.3	5,329.5	4,614.3	3,204.4
Lowest	493.0	392.0	380.0	349.0
Highest	28,185.0	21,407.0	23,194.0	14,956.0

	RICH/DISTRIBUTED	RICH/CENTRAL	LIGHT/DISTRIBUTED	LIGHT/CENTRAL
Event Relocation				
Messaging Time (ms)				
Message count	32.0	60.0	26.0	36.0
Total	6,445.0	39,646.0	4,749.0	9,249.0
Average	201.4	660.8	182.7	256.9
St. Dev	44.1	465.6	83.9	144.5
Lowest	93.0	115.0	113.0	123.0
Highest	279.0	2,077.0	350.0	559.0
Message Size (bytes)				
Message count	30.0	61.0	36.0	44.0
Total	352,563.0	714,129.0	4,666.0	38,200.0
Average	11,752.1	11,707.0	129.6	868.2
St. Dev	7,758.4	7,284.3	18.2	1,800.6
Lowest	2,661.0	2,704.0	116.0	116.0
Highest	32,995.0	29,612.0	186.0	8,744.0
Processing Time (ms)				
Message count	30.0	30.0	10.0	8.0
Total	61,306.0	74,708.0	51,990.0	34,522.0
Average	2,043.5	2,490.3	5,199.0	4,315.3
St. Dev	3,998.1	3,444.6	2,225.8	1,750.6
Lowest	0.0	0.0	3,224.0	2,586.0
Highest	17,734.0	13,708.0	11,395.0	8,458.0
Bulletin				
Messaging Time (ms)				
Message count	31.0	30.0	10.0	8.0
Total	9,401.0	8,919.0	1,881.0	1,182.0
Average	303.3	297.3	188.1	147.8
St. Dev	169.3	94.3	100.8	16.5
Lowest	156.0	153.0	99.0	129.0
Highest	929.0	519.0	458.0	174.0
Message Size (bytes)				
Message count	31.0	30.0	9.0	8.0
Total	413,319.0	376,211.0	47,718.0	33,716.0
Average	13,332.9	12,540.4	5,302.0	4,214.5
St. Dev	9,994.9	7,963.4	3,538.1	2,036.1
Lowest	2,723.0	2,766.0	2,374.0	2,374.0
Highest	42,711.0	29,612.0	13,514.0	8,744.0

	RICH/DISTRIBUTED	RICH/CENTRAL	LIGHT/DISTRIBUTED	LIGHT/CENTRAL
Processing Time (ms)				
Message count	28.0	28.0	39.0	39.0
Total	25,367.0	12,285.0	0.0	0.0
Average	906.0	438.8	0.0	0.0
St. Dev	1,258.7	572.6	0.0	0.0
Lowest	146.0	116.0	0.0	0.0
Highest	5,206.0	3,267.0	0.0	0.0
Summary				
Total Time (ms):	50,663,292.0	63,106,208.0	10,061,168.0	9,899,094.0
Overhead %:	2.3%	4.7%	5.4%	12.6%
Total Data (B):	3,732,542,350.0	6,876,953,343.0	31,220,779.0	140,040,547.0
Start time (from log)	15:23:58	9:16:29	14:37:07	11:25:55
End time (from log)	16:04:08	9:55:51	15:17:28	12:06:15
Total runtime:	0:40:10	0:39:22	0:40:21	0:40:20

Sandia National Laboratories

U.S. DEPARTMENT OF ENERGY