Proof of Concept Implementation of an Enterprise Service Bus for Health Information Exchanges

Philip Christian Zuniga, Joseph Benjamin Del Mundo, Edgardo Felizmenio, Marie Jo-anne Mendoza and Rose Ann Zuniga
Computer Security Group, Department of Computer Science, University of the Philippines - Diliman

Abstract – *Integration of health systems is one of the biggest problem in eHealth today. There are a lot of systems, yet they were developed using different platforms and technologies, making them virtually impossible to connect. In this paper, we discussed how to implement an ESB as the integration platform for health data. We identified use cases and functional requirements. Logical and deployment architecture were developed, and an actual proof of concept of an ESB is developed. Experiments were also done to determine the overhead caused by the ESB. Some of the functionalities of the ESB were examined to determine their individual overheads.*

Keywords: Enterprise Service Bus, Interoperability, Health Information Exchange, OpenHIE.

I. INTRODUCTION

During the past few years, the healthcare industry has witnessed a growth in the development of Health Information Systems. Hospitals, both public and private, rural health clinics, and even individual clinics of medical practitioners have been developing or acquiring their own Electronic Medical Records (EMRs), and Hospital Information Systems (HIS). The goal of these systems is to ensure a secure, organized and effective way of collecting and storing patient data [1]. The increase in the number of edge applications leads to the increase of patient data that are stored separately and independently, depending on the application that collected the data. The goal is to develop a Health Information Exchanges (HIE) [2][3]. An HIE is a system where various edge applications are allowed to share and access data that were collected by a different edge applications. However, HIEs are not easy to design, and eventually develop as we should consider the fact that the various POC applications run over varying platforms and technologies and use different terminologies (both medical and non-medical) and different business rules. The common software engineering solution to this kind of issue is to incorporate an interoperability layer (IL) in between the applications and the databases that handle patient medical data. This ensures that applications with different technologies can still communicate with each other seamlessly, and each edge application can pull out data from the databases even without knowing how the databases are configured. [5][6][7]. In this study, we will present the functional and non-functional requirements needed in setting up a Health Enterprise Service Bus implementation for the IL and the corresponding architectural design for such a middleware component. The OpenHIE framework will be used to simulate the implementation of a national HIE. In the next section, we will discuss the theoretical framework on our work, and a discussion of some of the previous works done in the area. We will then present the requirements for an Enterprise Service Bus under the context of healthcare. In the fourth section, we will discuss our architectural design. The benchmarking of ESBs will be discussed in the 5th section. We will discuss the Proof of Concept in the 6th section, while the experiments and the results will be discussed in the 7th section. We will summarize our results in the last section.

The following are the main contributions of this work:

1) Benchmarking framework for selection of ESBs in a health context
2) Design and conduct of experiments showing the effect of using an ESB in a health information exchange.

II. THEORETICAL FRAMEWORK

We will present in this section some of the theoretical requirements that are needed in our discussions throughout the paper. The purpose is to provide an overview that should be enough for a reader to understand the various concepts that will be discuss in the latter sections. The topics that will be discussed are:

1) Software Interoperability,
2) Health Information Exchanges
3) Open HIE
4) Enterprise Service Bus

I.1 Software Interoperability

The problem is that with the amount of applications developed on different platforms, using different protocols and standards, it will be very complex to connect them together pairwise. In a theoretical scenario of n applications needing to communicate with each other pairwise, there are around $O(n^2)$ bidirectional communication in the system. This creates complexity since each application will need to have $n - 1$ distinct interfaces that will need to be maintained. Changes in one of the applications or components will need to reflect to interfaces. This causes inefficiency in maintaining the system and will have a lot of cost overhead in the performance of the system. The idea of interoperability is for all the components to communicate with one another in an optimal way [8]. An interoperability layer is a middleware component that connects to all components of the system. It has the following basic functionalities:

1) Routing of messages
2) Validation of messages
3) Terminology Mapping
4) Mediation
5) Service Orchestration

For this work, we will be using an Enterprise Service Bus (ESB) as our interoperability layer. An ESB is a monolithic application that has various modules inside it. The difference of an ESB over other interoperability layer implementation is that an ESB is a single application. It is always used in conjunction with a Service-Oriented architecture supported system. ESBs are also used in large enterprise type system. In a service-oriented architecture, the goal is to always avoid point to point integration (Figure 1).

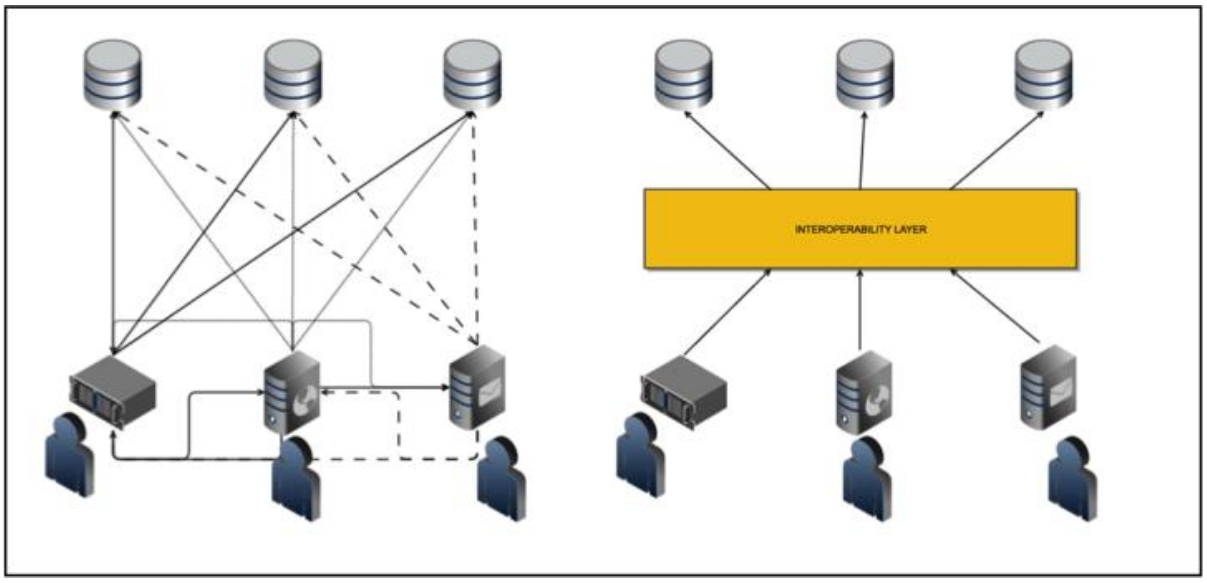


Figure 1. Point to point integration versus using an Interoperability Layer

I.2 Health Information Exchange Implementations

There are several ways to implement an HIE: a (1) Centralized Approach, (2) Federal Approach and (3) A Hybrid approach. Though we will discuss each approach briefly our focus is on Open HIE framework, which will be discussed more in detail later.

In a centralized architecture (Figure 2), the HIE will have a centralized repository for health data. Access protocols will be defined by a central authority, which is usually governed by a countrys MOH. Though a centralized approach can work on the national level, it can be implemented on a regional or local level. Hospitals and other point of care institutes would submit their health data to the database, and at the same time, they should be able to pull data from the same centralized database. Patient data flow and updating of data can easily be done since there is only a single database that needs to be updated.

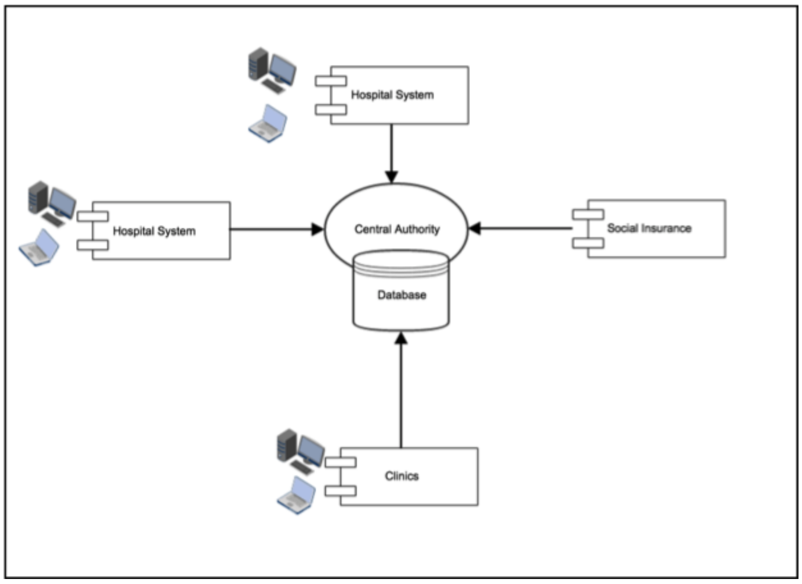


Figure 2. Centralized HIE

On the other hand in Federated Architecture (Figure 3), there will be different repositories for each of the locality. Patient data will not be stored together. Data exchange between two repositories of different location will need to pass through the national authority. A Federated architecture tend to be less interoperable than a centralized system since message passes through at least two channels.

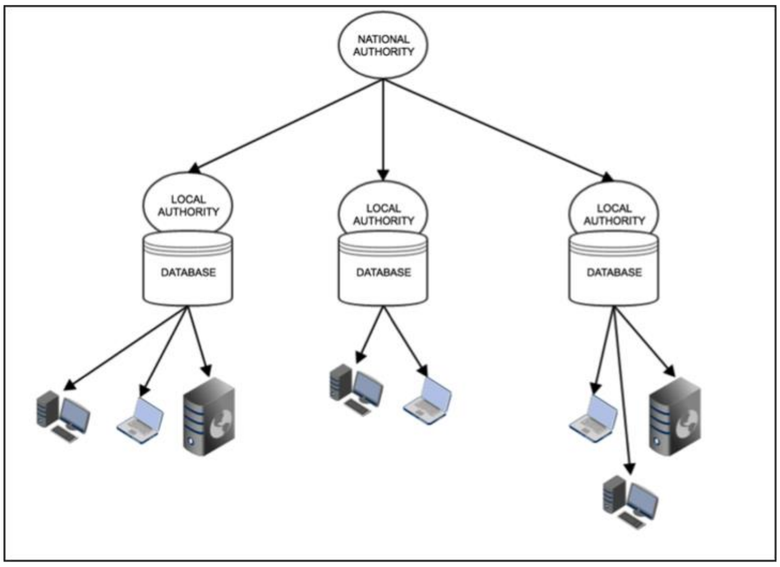


Figure 3. Federated HIE

In a hybrid model, there are local database in the localities, and a national database that mirrors the content of the local databases. It has the same design as the federated HIE except for the additional national database. Recently more modern approaches to HIEs have been developed, which are more focused on interoperability rather than on the hierarchy of access. We present now Open HIE (OHIE). OHIE will be basis of our design and requirements.

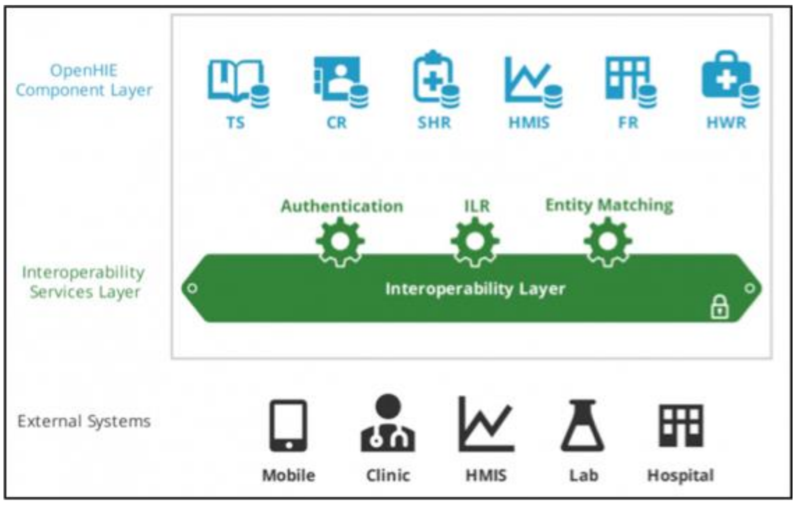


Figure 4. OpenHIE Framework

I.3 OpenHIE (OHIE)

OpenHIE (www.openhie.org) has been designed to promote shared databases, reusable services and interoperability between different components. As compared to other architectures, which assume seamless communication among the different HIE components, such an assumption is not used in OpenHIE.

As we can see in Figure 4, OHIE promotes the use of shared databases. The design of the architecture is such that there will be a:

- Client/Patient Registry: Database that contains data on all possible patients in the locality.
- Facility Registry: Database that contains data on all health providing facilities in the locality.
- Health Worker Registry: Database that contains data on all the health workers in the locality. This includes the list of doctors, nurses, lab operators etc.
- Terminology services: This is a mapping of terminologies used by the different external systems.
- Health Management Information System: This is the central module for access to health data. It may include an executive dashboard.

External systems that need to connect to the different databases include point of care applications/edge systems (both computer-based or mobile-based), laboratory systems, pharmaceuticals and social insurance systems. In our case we limit out discussion to point of care applications only.

In between the external systems and the OHIE components is the interoperability layer. The interoperability layer ensures that systems using different technologies and platform can still communicate. The interoperability layer also lessens the complexity of managing multiple applications that communicates with one another pairwise. There are a lot of possible architecture and framework for the interoperability layer, but we are proposing the use of an enterprise service bus (ESB) as our interoperability layer. The OpenHIE framework is being used as it is and provides an open framework, it supports a service-oriented approach and it clearly specifies the interoperability layer.

I.4 Enterprise Service Bus

An ESB is an architectural framework. It defines a set of rules, modules and protocols on how different applications can be integrated in a bus like environment. A bus like environment implies that there is no data that is left in the ESB, as it has no memory of its own that can handle data. The core concept of an ESB is that you set up an interoperability layer, with the ESB inside it, and different applications can communicate with one another by communicating with the ESB. Aside from being a communication platform, the ESB can also authenticate, validate, and transform messages. The ESB can also connect [4] to an external terminology services to have terminology translation. We obtain the following advantages with the use of an ESB:

- Better management of web services
- Changes in a component do not lead to change in other components.
- Single authentication protocol
- Legacy systems can still be used.
- Errors in communication will not propagate.

III. REQUIREMENTS FOR AN ESB

We now present in this section a discussion on the requirements for a Health Enterprise Service Bus for an Open HIE type HIE. The requirements though may be requirements for any generic implementation of an ESB is more focused towards the consideration that the ESB will be implemented over an HIE. There will be focus on data security and privacy, data validation and data translation. We only focus on the functional requirements, as the distinction of using an HESB is more obvious in the functional requirements and not in the non-functional requirements.

I.1 Use Cases

Before presenting the actual requirements, we first discuss the use cases of an ESB. Take note that as a software component, the ESB has no public interface except for the web admin page. An ESBs interface with the other components is defined by services/APIs that it will expose to the other components. In the use cases that will be shown below, most of the use cases are defined, where the edge application (EMR/HIS) will be the actor. We also look at use cases where the OpenHIE components (registries) are the actors. The following are the use cases:

1) Add Medical Data: An edge application sends a medical data (regardless of the type) to the ESB. The ESB will process the data and determine which registry will the data must be sent
2) Look up patient data: An edge application sends a patient ID that will be used by the ESB to look up for a particular patient in the HIE Components. The HIE components will respond if the particular patient exists, and will send the data to the ESB, then the ESB will send the data to the edge application.
3) Look up medical record: An edge application sends a medical record ID that will be used by the ESB to look up for a particular medical record in the HIE Components. The HIE components will respond if the particular patient exists, and will send the data to the ESB, then the ESB will send the data to the edge application
4) Send Response: The registries will send their response to the ESB.
5) Send patient data: The registry will send patient data to the ESB
6) Send patient medical record: The registry will send medical record to the ESB.
7) Send health provider data: The registry will send health provider data to the ESB.
8) Send health facility data: The registry will send health facility data to the ESB.

The functionalities discussed in this section are baseline functionalities for an ESB if it will be implemented on an HIE. The functionalities are not particular to OHIE, though the design on how they were listed assumes OHIE as the architecture being used by a particular HIE.

I.2 Functional Requirements

From the identified use cases, we now discuss the resulting functional requirements of the ESB. These functional requirements are the essential requirements of the ESB, particularly on the health context.

1) Connection Authentication: Data sent to the ESB by the edge application should be encrypted, as medical data should always be secured because it contains sensitive information about a person. Symmetric protocols may be used in authentication while public key cryptography can be used for key exchange.

2) Content based Routing: There should only be a single data interface between the ESB and the various edge applications. Any kind of data can be sent by the edge application through the interface, and it will be the ESB that will decide, based on the content of the data, on where to route the data. This is an important functionality, as this will promote efficiency on the processes because only a single interface will be used.
3) Data Validation: It is important that data validation be included in the major functionalities, since this will prevent erroneous data from being sent to the registry and with a possible propagation error. Data validation will check on the input schema against the actual schema of the registries.
4) Data Transformation: Due to the possible difference in data formats available and are being used by the different edge applications, Data transformation is an important module because the ESB should be able to transform data from one format to another and vice versa.
5) Data Translation: The ESB should have an internal or external terminology services that can map terminologies used by different edge applications. This is important particularly for medical data, as there is no single standard for the terminologies.
6) Queuing: Queuing of request will be crucial especially in high traffic scenarios.
7) Business Intelligence: The ESB must have some form of Business Intelligence (BI) to help it in lessening the load of the different components connected. A possible BI implementation is for the ESB to generate data/report based from a submitted data to it, and submit it to a particular registry. This makes the ESB able to make itself not only as an integrator but as an orchestrator too.

The functionalities discussed in this section are baseline functionalities for an ESB. The functionalities are not particular to OHIE, though the design on how they were listed assumes OHIE as the architecture being used by a particular HIE.

IV. TECHNICAL ARCHITECTURE

In this section we present the logical architecture, its corresponding logical workflow and deployment architecture of the ESB, implemented over OHIE. The logical architecture presents how the different modules/functionalities will fit in an ESB, while the deployment architecture visualizes the appearance of how the ESB will be deployed.

Figure 5 shows the proposed logical architecture of the HESB. We integrate in the architecture the various logical modules of the ESB. The queuing module is the first module that will be encountered. Once a server can take a request, it will authenticate the received request. Various data handling mechanisms are now available if the request was authenticated to be coming from an authenticated edge application. Among the data handling mechanisms are: validation, transformation, translation and content based routing. Take note that the translation module is connected to an external terminology service. Some ESBs already have its own native terminology services.

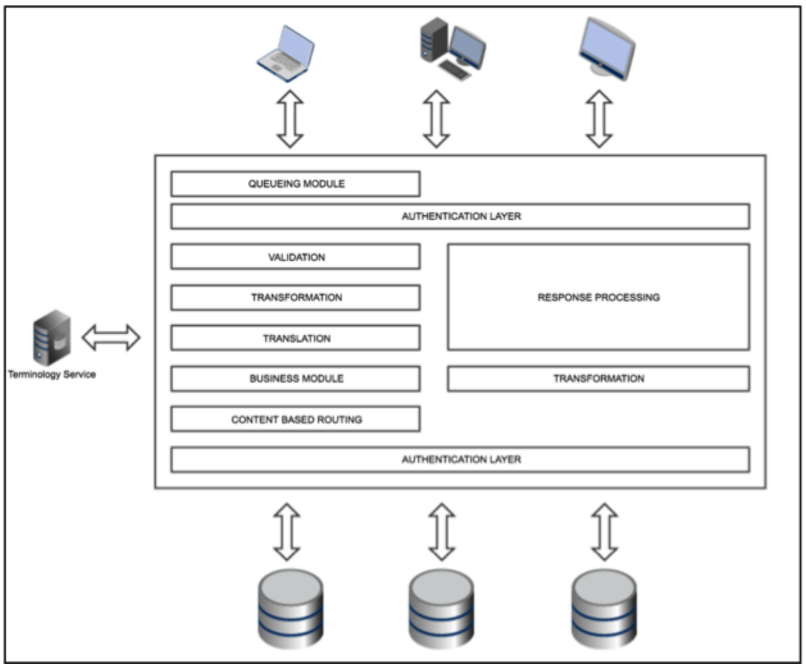


Figure 5. Logical Architecture

The last layer is another authentication module before the data is sent to the registries. If the registries will be hosted to another secured server, then the last authentication module may no longer be used. Since the responses of the registries are controlled it will undergo less processing than when messages/data are sent from the edge systems to the registries. Fig 6 and 7 shows a workflow perspective to the logical architecture presented in the earlier figure. In the logical workflow we can see an ordered sequence of the modules as they are being used by the ESB, There is a difference between the workflow when the data comes from the edge systems to the registries or when the registry is sending response to the edge application.

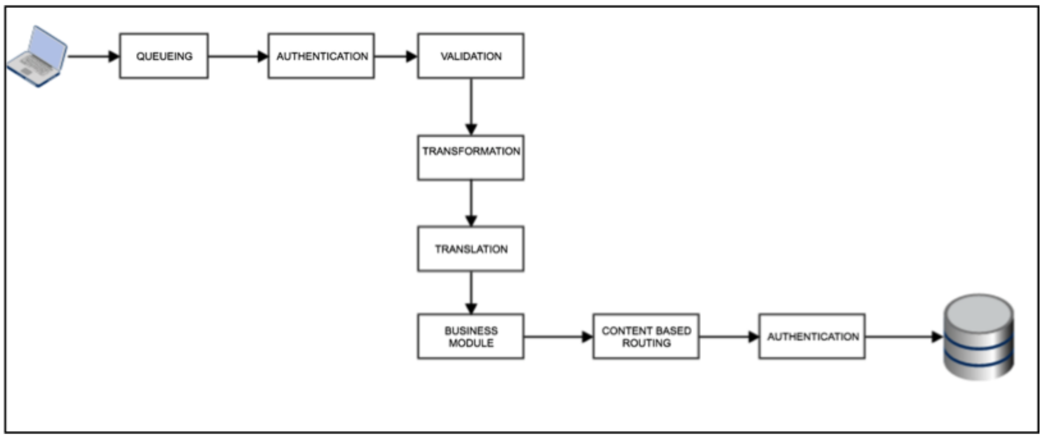


Figure 6. Logical Architecture (Edge applications to Registries)
Figure 7.

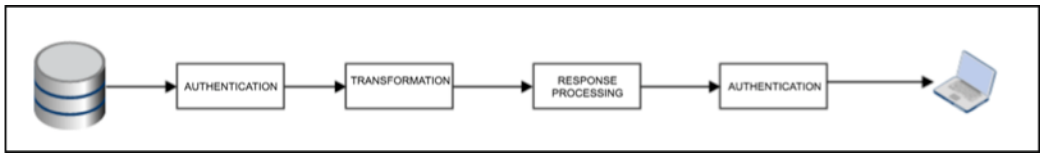


Figure 8. Logical Architecture (Registries to Edge applications)

Logical Workflow Steps (Edge application to Registries)

1) Edge application sends request/message to the ESB
2) The message queues at the Queuing Module
3) The message is authenticated
4) The message is validated for correct format and schema
5) The message is transformed based on the required format of the receiving registry.
6) The message is translated via an external terminology services.
7) If some business rules need to be applied to the message, then it will be applied at this point.
8) The ESB will determine where the message will be sent.
9) The message will be encrypted if needed.
10) The message will be sent to the registry.

Upon reply of the registries to the edge application we will have the following workflow:

Logical Workflow (Registries to edge Applications)
1) The registry will send response to the ESB
2) The registry's response will be authenticated
3) The registry's response will be formatted based on the required format of the edge application.
4) Other responses may be included
5) The response will be encrypted
6) The encrypted response will be sent to the edge application.

We also present the deployment architecture of an ESB. The deployment architecture shows the set-up of how a system can be deployed.

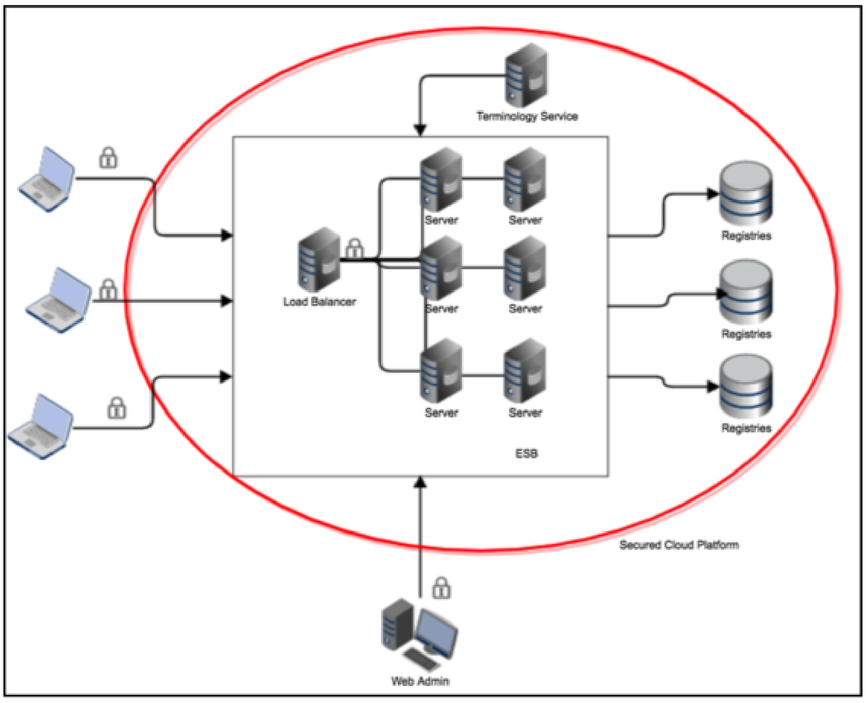


Figure 9. Deployment Architecture

V. SELECTION OF ESB FOR PROOF OF CONCEPT

As part of the proof of concept development, we made a selection over several proprietary and open source enterprise service buses. It is important to mention that our benchmarking is a desktop benchmarking as we didn't have the time to do an actual feature evaluation of the ESB. The following are the criteria together with their assigned weights that were considered for this research:

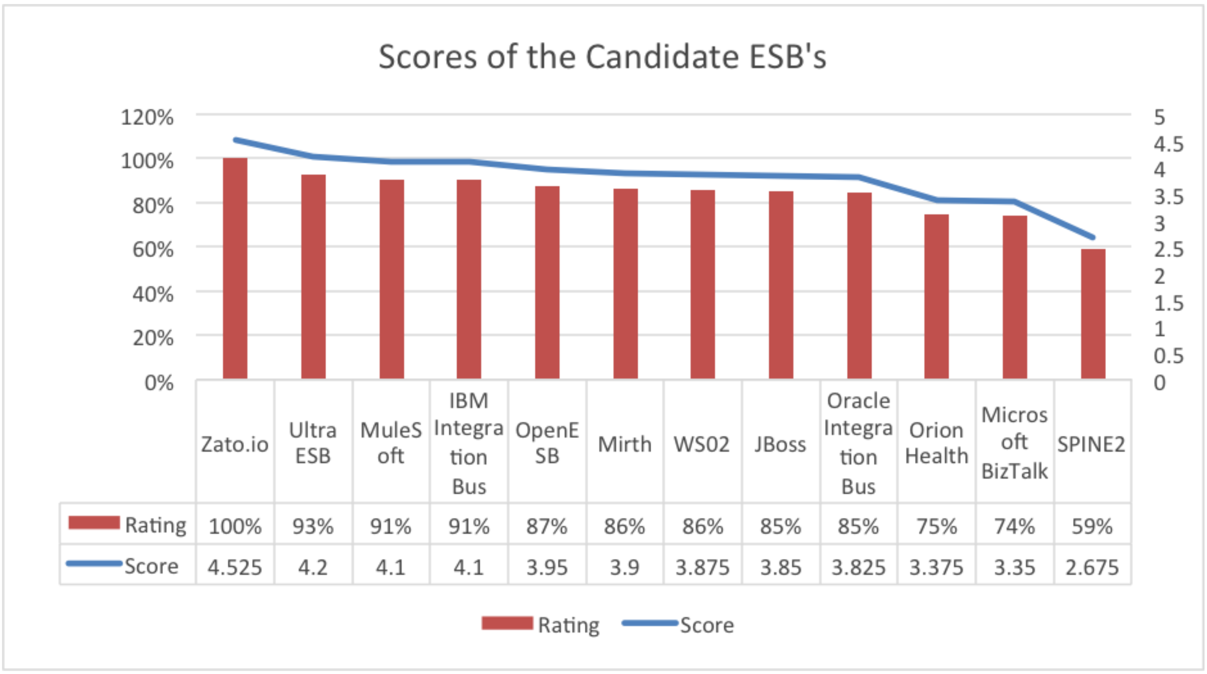
1) Ease of Installation (5%)
2) Dependencies (7 %)
3) Ease of Coding (18 %)
4) User Support and Documentation (11 %)
5) Scalability (15 %)
6) Cross Platform (4 %)
7) Security (15 %)
8) Sustainability (15 %)
9) Testimonials (2 %)
10) Support to Open Source Technologies (9 %)

The weights were obtained via a round robin method, where each criteria is compared to another criteria to get a point. The points are then added and normalized to obtain the necessary weights. The following ESB's were considered for this work:

1) IBM Integration Bus (http://www-03.ibm.com/ software/products/en/ibm-integration-bus): The IBM integration bus is part of IBMs wide list of IT products. As an integration layer, it offers a fast and simple way for systems and applications to communicate with each other by using standard IBM technologies.
2) Microsoft BizTalk (https://www.microsoft.com/en-us/ cloud-platform/biztalk): Microsoft BizTalk is a collection of tools and libraries that extend BizTalk Server 2010 capabilities of supporting a loosely coupled and dynamic messaging architecture. It functions as middleware that provides tools for rapid mediation between services and their consumers. Enabling maximum flexibility at run time, the BizTalk ESB Toolkit 2.1 simplifies loosely coupled composition of service endpoints and management of service interactions.
3) UltraESB (https://www.adroitlogic.com/products/ ultraesb/): UltraESB is a Free and Open Source Enterprise Service Bus, first released in January 2010 under the AdroitLogic Zero-Dollar EULA. Since August 2010 its source code has been made available under the OSI approved Affero General Public License (AGPL) version 3. UltraESB is in production use at many leading organizations around the world, and its source code is available under custom licenses for larger enterprise deployments. It has been licensed under the terms of an customized Apache Software Foundation License Version 2, by one of the largest organizations in the world among the Fortune 10
4) SPINE2 (https://nhsconnect.github.io/spine-eis-basics/ develop spine two.html): An upgrade from the old SPINE system used by the NHS. This system uses open source technologies, but the license of the ESB is not open source. The system uses Riak, Nginx, Tornado, Redis and RabbitMQ
5) Orion Health (https://orionhealth.com/): Orion Health is a full suite software solution for health workflow. As an integration layer, Orion Health uses a Rhapsody integration bus to provide interoperability across systems of different platforms.

6) Oracle Integration Bus (http://www.oracle.com/us/products/applications/061928.html): Oracle ESB is technically an enterprise service bus designed and implemented in an Oracle Fusion Architectures SOA environment; to simplify the interaction and communication between existing Oracle products, third-party applications, or any combination of these. As a software architecture model for distributed computing it is a specialty variant of the more general client server software architecture model and promotes strictly asynchronous message oriented design for communication and interaction between applications. Its primary use is in Enterprise Application Integration of heterogeneous and complex landscapes of an organization, and thus enabling its easy management.
7) WS02 (http://wso2.com/): WSO2 Enterprise Service Bus is a lightweight, high performance, near-zero latency product, providing comprehensive support for several different technologies like SOAP, WS* and REST as well as domain-specific solutions and protocols like SAP or HL7. It goes above and beyond by being 100% compliant with enterprise integration patterns
8) Mirth (https://www.mirth.com/): A cross platform ESB that is used in health applications. Its most useful sub module, Mirth Connect is an HL7 engine that enables bi-conditional sending of HL7 messages, among different systems. Mirth is built over Java.
9) MuleSoft (https://www.mulesoft.com/): The Mulesoft ESB is an integration platform for connecting enterprise applications on-premises and to the cloud, designed to eliminate the need for custom point-to-point integration code. It uses Java and runs across all platforms.
10) JBoss (http://www.jboss.org/): The JBoss Enterprise SOA Platform is free software/open-source Java EEbased service-oriented architecture (SOA) software. The JBoss Enterprise SOA Platform is part of the JBoss Enterprise Middleware portfolio of software. The JBoss Enterprise SOA Platform enables enterprises to integrate services, handle business events, and automate business processes, linking IT resources, data, services and applications. Because it is Java-based, the JBoss application server operates cross-platform: usable on any operating system that supports Java. The JBoss SOA Platform was developed by JBoss, now a division of Red Hat.
11) OpenESB (http://www.open-esb.net/): OpenESB is a Java-based open source enterprise service bus. It can be used as a platform for both enterprise application integration and service-oriented architecture. OpenESB allows you to integrate legacy systems, external and internal partners and new development in your Business Process. OpenESB is the unique open-source ESB relying on standard JBI (Java Business Integration), XML, XML Schema, WSDL, BPEL and Composite application that provides you with simplicity, efficiency, long-term durability, and savings on your present and future investments with a very low TCO (Total Cost of Ownership).
12) Zato.io (https://zato.io/): Zato is an ESB and application server written in Python and can be used for building middleware and backend systems. It is opensource software with commercial and community support available. Python is a programming language famous for its ease of use and productivity.

Each of the ESB were benchmarked based on the criteria specified above and we got the following results (Figure 9).



	Zato.io	Ultra ESB	MuleSoft	IBM Integration Bus	OpenESB	Mirth	WS02	JBoss	Oracle Integration Bus	Orion Health	Microsoft BizTalk	SPINE2
🟥 Rating	100%	93%	91%	91%	87%	86%	86%	85%	85%	75%	74%	59%
― Score	4.525	4.2	4.1	4.1	3.95	3.9	3.875	3.85	3.825	3.375	3.35	2.675

Figure 10. Results of Benchmarking

Zato.io got the highest score mainly because it uses Python, hence it was easier to code using it as compared to other ESBs which were mainly written in Enterprise Level Java. Zato.io also scored high in sustainability and openness due to the fact that it is an open source technology.

VI. PROOF OF CONCEPT

Zato.io is used as the interoperability layer for this research. The proof of concept (POC) that was developed aims to show the following:

- It will enable various edge systems to connect to one another
- An ESB has low overhead costs.
- It has high scalability.
- It has fault tolerance properties.
- Identify the workflow improvements that can be obtained when using an ESB.

The following figure (Figure 10) is the set up that was followed in the proof of concept. In the proof of concept, an edge system may choose to push patient record (in JSON) to the registries by passing through the ESB. It may also pull a patient record (with a corresponding ID). The registry interfaces has pull/push API's that can be accessed by the ESB.

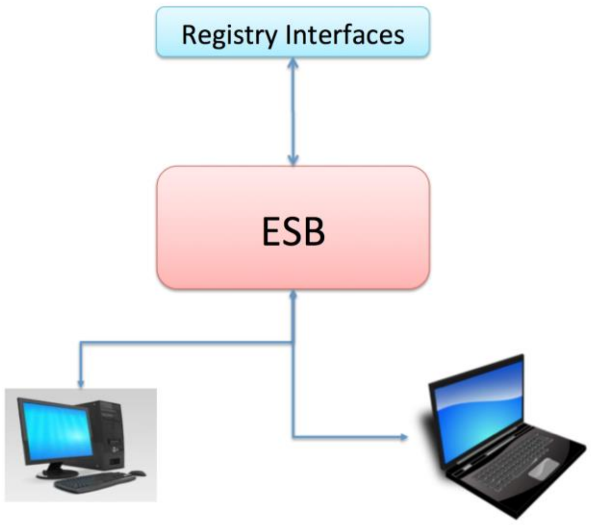


Figure 11. Proof of Concept diagram

In the presented proof of concept, queueing, routing, authentication, data transformation and validation are done in the ESB. The same functionalities are also available in the registry interfaces for point to point integration.

It is also important to note that the ESB will be placed in a cluster of computers. In the proof of concept, the ESB is deployed in a cluster of two servers, with a separate load balancer (also included in the Zato.io) installation. The use cases and functionalities identified in the earlier section were implemented in the proof of concept. The following tests were done to check the functionalities of the ESB.

1) *Queueing Tests*: Multiple and concurrent requests were sent to the ESB using SoapUI. The logs of the ESB has shown how the requests queued at the load balancer before they were assigned to a particular cluster. This demonstrates how queuing is done in the load balancer.
2) *Authentication Tests*: A public/private key pair is provided by the ESB to the various edge systems connecting to it. A successful authentication process was checked by sending a valid key pair to the ESB. Another test that was done is when the edge system sends a wrong key pair to the ESB and the ESB should reject such a connection. Both tests were satisfied by the ESB.
3) *Validation*: ESB validation is tested by sending a valid data/format to the ESB and the ESB should be able to return a “Valid data response”. Also, a malformed data was sent to the ESB to check for false positives. Sending a malformed data to the ESB should trigger an “Invalid data response” from the ESB. Both cases were also tested.
4) *Transformation*: The ESB should be able to format the data to make the data compatible to the recipient’s data format. In this test, the data format of the receiving interface was toggled between an XML format and a JSON format. The Edge systems both submitted a JSON formatted file. The ESB was able to convert the data to XML format if XML format is needed.

5) *Routing*: Routing was tested by checking if the correct recipient receives the correct data even if it was specified by the sending system.
6) *Fault Tolerance*: Since the ESB is deployed over a cluster with a load balancer, once a server fails the load balancer automatically reassigns the requests assigned to it to another server so that the processing of the request continues even after server failure.

All the tests were satisfied by the ESB during the POC implementation.

VII. RESULTS AND ANALYSIS

Aside from testing the functionalities, several quantitative tests were done to measure the effects of using an Enterprise Service Bus in an HIE. Mostly, transaction/processing time was measured as the main parameter of the tests. Transaction time is defined as time between sending a request from an edge system to the point where the same edge system will receive a response. The following are the experiments that were done:

7.1 Experiments and Results

1) *ESB overhead Costs*: The cost of sending a message to the ESB instead of sending it directly to the interface systems was measured by computing for the transaction time when an edge system sends to the ESB a message against when the edge system sends the data directly to the registry interface. The difference in the transaction time shows the overhead cost produced by the ESB.

   Figure 11 shows the difference in transaction time when data is sent to the ESB against if the data is sent to the registry interface directly. The overhead varies from payload size. It is increasing as the payload size increases. However, if the payload size is < 20kb, then the overhead size should be at a manageable range of < 5%.

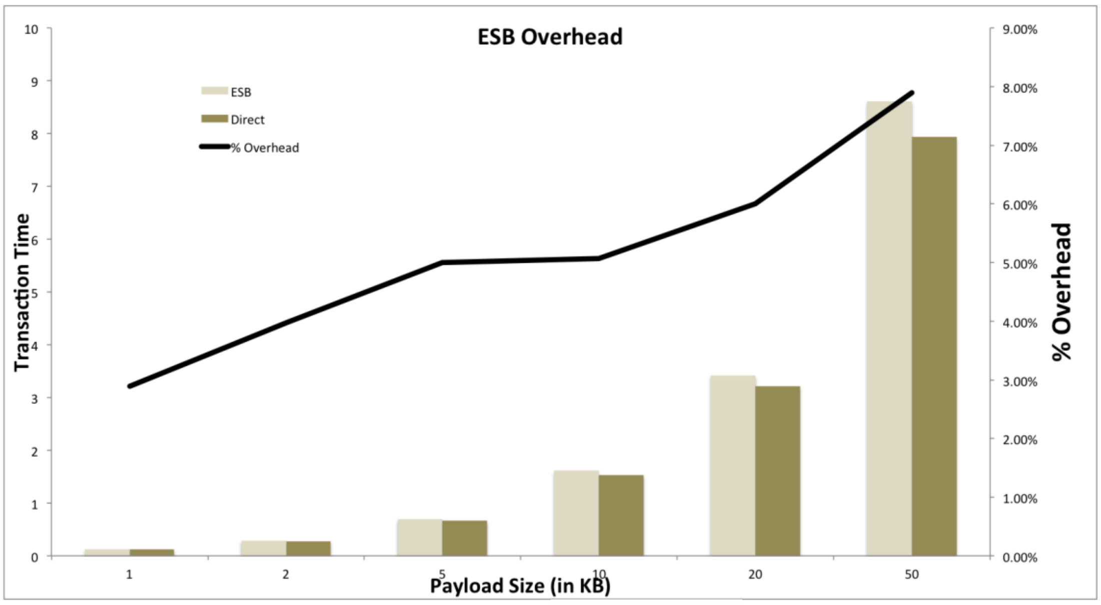


Figure 12. ESB Overhead

2) *Effect of Payload Sizes*: The graph shown in Figure 11 also shows that transaction time increases linearly as the payload size increases. This is totally expected since large data should have larger transaction time.
3) *Effect of Authentication*: The effect of authentication in ESB processing was measured by recording the total processing time with authentication, and the total ESB processing time without authentication.

   The result of the test as shown in Figure 12 indicates that the overhead due to authentication increases as the payload size increases. This is expected since it will take a larger file longer to be encrypted as compared to a smaller file. The range is quite significant for large files, while it can be ignored for small files.

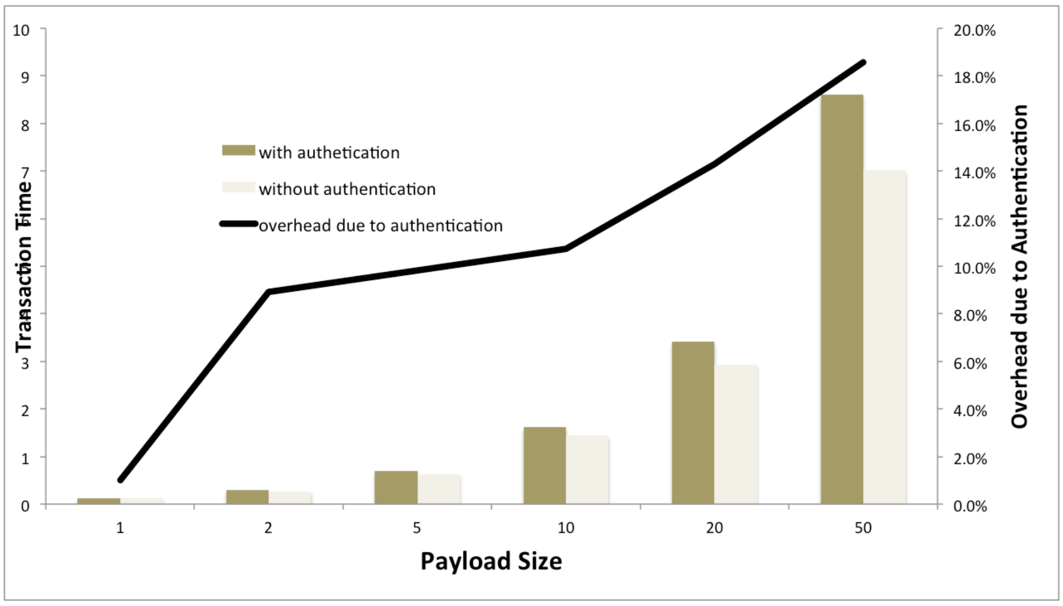


Figure 13. ESB Authentication Overhead

4) *Effect of Validation*: Schema based validation is expected to be the most costly process in an ESB as it will check the whole formulation of the data that will be sent. Validation is important because it will ensure that no malformed data is sent to the ESB's data center. Just like in other experiments, varying payload sizes were used, and the validation property was tangled between ON and OFF The result of the experiment shows that validation tends to have an increasing effect with an increase in payload sizes. Generally, the overhead due to validation increases as payload size increases. However, the overhead tends to flatten out for larger file sizes.

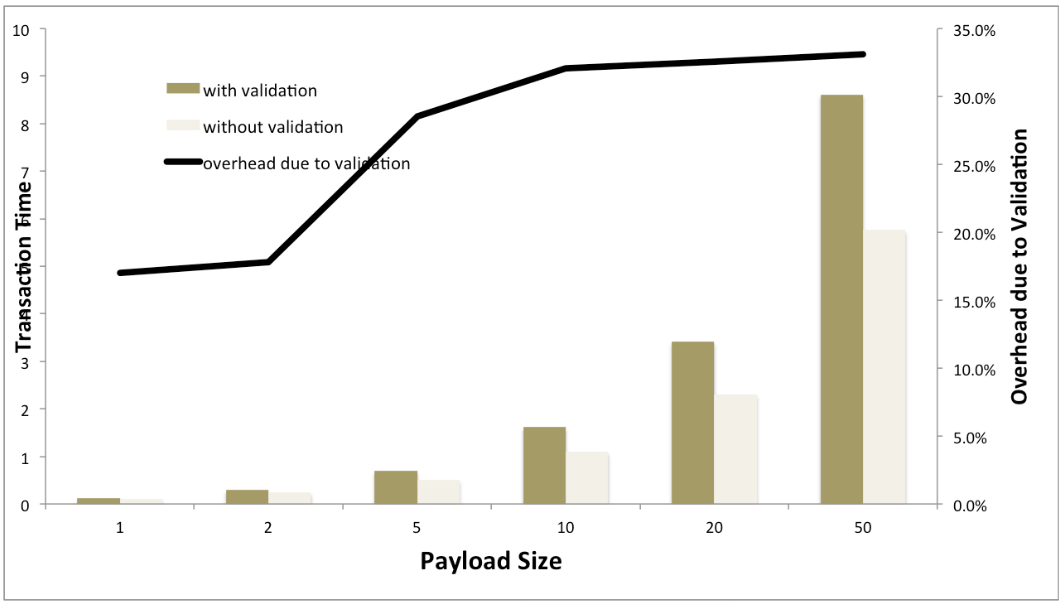


Figure 14. ESB Validation

5) *Queuing Overhead*: The effect of queuing was measured by sending concurrent request to the ESB using SoapUI (https://www.soapui.org/). A range of concurrent requests were used (5, 10, 15, 20, 25). The processing time and the ESB processing time was measured. The processing time is the time when the edge system sent a request to the time it received a response. The ESB processing time is the time an ESB received a request to the time the request went of the ESB to be sent to the registry interface.

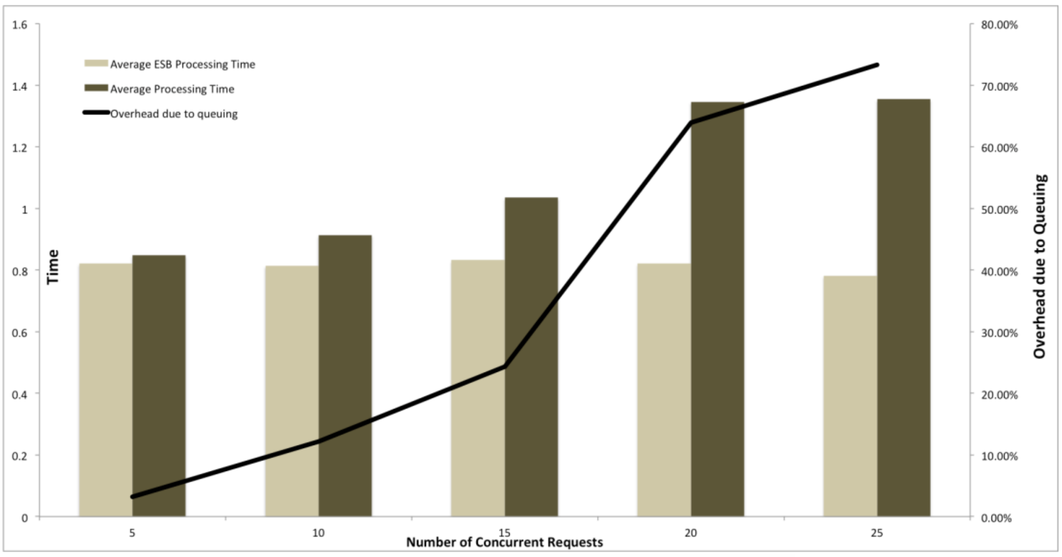


Figure 15. ESB Queuing of Requests

The result shows that as the number of concurrent request increases, the effect of queuing becomes higher as the requests tend to wait at the load balancer longer when there are longer queues. It is important to take note that the ESB processing time tends to be constant regardless on the number of concurrent requests.

7.2 Advantages of using an ESB over Point to Point Integration

Aside from the quantifiable results that can be seen when using an ESB, we will also enumerate in this section a number of items that were observed during the implementation of the Proof of Concept.

1) *Content based routing*: The ESB can be used to route data based on its content. This different from point to point integration as, it should identified where to route data before it is sent.
2) *Lower Connection Complexity*: In point to point integration, the maximum number of connections for a system with n components is $\frac{n(n-1)}{2}$, or *O(n²)*. This can be reduced to *O(n)* when an ESB is used.
3) *Propagation of Changes*: In case a component makes any changes in interfaces, all the other $n-1$ components would need to adjust too for a point to point system. If an ESB is used, only the ESB will be adjusted.
4) *Message Rules*: In point to point systems, an edge system can only send a particular message to a particular registry. If an ESB is used, an edge system can sent an aggregated message to the ESB and the ESB will decide which part of the message will be sent to a particular registry.
5) *Processing of Responses*: In point to point systems, an edge system would receive multiple responses if it sends data to multiple registries. If an ESB is used, responses can be consolidated at the ESB hence, a single response will received by the edge systems.

VIII. CONCLUSIONS

We present in this work a proof of concept implementation of an Enterprise Service Bus for Health Information Exchanges. The following are the activities that were done.

1) The use cases identified are the basic use cases used in health information exchanges.
2) A list of functional requirements were identified based on the use cases.
3) The OpenHIE framework is used. This framework is used because it supports a Service oriented approach, it is open, and it has an inherent interoperability layer in the framework.
4) Logical and Deployment architecture were developed to serve as basis of the implementation.
5) A list of several ESBs were benchmarked to identify which could be used for the proof of concept.

The following results were obtained:

1) Using an ESB provides a lot of advantages over point to point systems. Most of the advantages lies on the ease of managing the HIE with an ESB rather than point to point systems.
2) An ESB provides minimal overhead (< 5%) for small data sizes.
3) The processing time increases as payload size increases.
4) Validation provides a significant overhead. It is recommended that for complex validation, these can be done on the registry level.

5) Concurrent requests does not have an effect on ESB processing time, however multiple concurrent requests have an effect on the processing time and it increases as the number of concurrent requests increases. This can be managed if there are more servers that are included in the ESB cluster.

References

[1] H.D Masethe, O.O Olugbara et. Al, *Integration of Health Data using Enterprise Service Bus*, Proceedings of World Congress of Engineering and Computer Science 2013, vol II.

[2] Y. Gong, *Healthcare Information Integration and Shared Platform Based on Service-Oriented Architectures*, in 2nd International Conference on Signal Processing Systems (ICSPS), 2010, pp. 523-527

[3] N. D. Zuma, White Paper for the transformation of the Health System in South Africa, Pretoria, South Africa, 1994.

[4] A.Ryan, P.Eklund, The Health Service Bus: An Architecture and Case Study in Achieving Interoperability in Healthcare, MEDINFO 2010

[5] A.J. Ryan and P.W.Eklund, *Ontology Mapping between HL7 Versions 2 and 3 and OpenEHR for Observations Messages*. Proceedings of the Health Informatics Conference, 2009, HISA pp. 40-46, 2009.

[6] P. Taylor, From Patient Data to Medical Knowledge: the principles and practices of Health Informatics. BMJ Books. Blackwell Publishing Ltd. 2006.

[7] O. Gul, M. Al-qutayri, Q. H. Vu, and C. Y. Yeun, *Data Integration of Electronic Health Records using Artificial Neural Networks* in the 7th International Conference for Internet Technology and Secured Transactions, 2012, vol. 7, pp. 313-317.

[8] K.Ketaki, Data Integration of in Reporting Systems using Enterprise Service Bus Ohio State University, 2009

[9] D. P. Hansen, C. Pang, and A. Maeder, *HDI: integrating health data and tools*, Soft Comput, vol. 11, pp. 361-367, 2007.

[10] B. Smith, A. Austin, M. Brown, J. King, J. Lankford, A. Meneely, and L. Williams, Challenges for Protecting the Privacy of Health Information : Required Certification Can Leave Common Vulnerabilities Undetected, pp. 112, 2010.

[11] A. Dogac, *Interoperability in eHealth Systems*, no. July, pp. 2026-2027, 2008.

[12] OpenHIE.org: *https://ohie.org/*, lass accessed on March 22, 2017

[13] IBM Integration Bus: http://www-03.ibm.com/software/products/en/ibm-integration-bus, last accessed on March 22, 2017

[14] Microsoft BizTalk: https://www.microsoft.com/en-us/cloudplatform/biztalk, last accessed on March 22, 2017

[15] Ultra ESB: https://www.adroitlogic.com/products/ultraesb/, last accessed on March 22, 2017

[16] Spine2: https://nhsconnect.github.io/spine-eis-basics, last accessed on March 22, 2017

[17] Oracle: http://www.oracle.com/us/products/applications/061928.html, last accessed on March 22, 2017

[18] Orion: *https://orionhealth.com/*, last accessed on March 22, 2017

[19] WSO2: *http://wso2.com/*, last accessed on March 22, 2017

[20] JBoss: *http://www.jboss.org/*, last accessed on March 22, 2017

[21] OpenESB: *http://www.open-esb.net/*, last accessed on March 22, 2017

[22] Zato.io: *https://zato.io/*, last accessed on March 22, 2017

[23] Mulesoft:*https://www.mulesoft.com/*, last accessed on March 22, 2017

[24] Mirth: https://www.mirth.com/, lass accessed on March 22, 2017