Usable, lightweight and secure, architecture and programming interface for integration of Wireless Sensor Network to the Cloud.

THESIS

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Sharada Krishna Patil

Graduate Program in Computer Science and Engineering

The Ohio State University

2011

Thesis Committee:

Rajiv Ramnath, Advisor

Jay Ramanathan, Advisor
Abstract

Wireless sensor networks (WSN) have been gaining popularity in recent years because of their potential to enable innovative applications in various fields. These fields include industrial, social, and regulatory applications, to name a few. If we extend a traditional sensor network to the Internet, WSNs that are dispersed and networked together can collaborate to accomplish many tasks that cannot be accomplished with a few powerful sensors or computers on a smaller network. With the gaining popularity of cloud services due to the pay per use policy of computation and data storage resources and easing of the burden of maintaining the service, using the cloud to integrate a WSN to the Internet has become viable.

The primary goal of this research is to investigate how to facilitate secure communication between a WSN and a cloud and to provide a secure policy for users to access such a service. We propose and develop an architecture and programming interface, named as Intortus, to enable this exploration. Intortus lets software programmers develop and deploy applications on WSN quickly by relieving the programmer from understanding and using the cloud provider API to access its data store, and writing embedded C code for sensors. There are many challenges enabling such a service that provides end to end secure communication, design constraints due to limitations in the services of cloud providers to mention a few. We discuss the issues and approach taken to build such an architecture and interface. This thesis also describes the functionality Intortus provides and how it can be further extended.
Dedication

I dedicate this work to my parents, Radha and Krishna Patil, in laws, Yashoda and D.T. Rangarajan and my husband, Nishchal Devanur for their constant support. I dedicate this thesis, as well, to my brother and sister in law, Ramesh Patil and Uma Ravi for helping me out in providing me suggestions and advice on technologies and design approaches.
Acknowledgments

I thank my teammates and friends, Nived Sivadas and Kartik Natarajan, for their help with developing the Intortus firmware and making my thesis a joyful ride. I am also grateful to my advisors, Rajiv Ramnath and Jay Ramanathan, as well as the Samraksh Company for supporting me throughout my Master’s program.
Vita

May 2008 ........................................... ............B.Tech. Computer Engineering,

National Institute of Technology, Karnataka, India

September 2009 to present.........................Graduate Student, Department of Computer Science and Engineering,
The Ohio State University

Fields of Study

Major Field: Computer Science and Engineering
# Table of Contents

Abstract .............................................................................................................................................. ii

Dedication ........................................................................................................................................ iii

Acknowledgments ............................................................................................................................ iv

Vita .................................................................................................................................................... v

Table of Contents ............................................................................................................................. vi

List of Figures .................................................................................................................................. viii

Chapter 1: Introduction .................................................................................................................... 1

Chapter 2: Cyber Infrastructure Application Scenarios ................................................................. 6

Chapter 3: Related Work .................................................................................................................. 9

  3.1 Cloud computing solution for health care data collection......................................................... 9

  3.2 Mobile Body Sensor Network for Health Applications ............................................................ 11

  3.3 A framework for sensor cloud integration: Opportunities and Challenges ......................... 12

  3.4 TinyDB .................................................................................................................................. 13

  3.5 Secured WSN-integrated Cloud Computing for u-Life Care (SC3) ....................................... 14

Chapter 4: Architecture and Programming Interface ..................................................................... 16

  4.1 Sensor Node and Gateway ....................................................................................................... 19
4.2 The Cloud and Data Store ................................................................. 20

4.3 Security model .................................................................................. 22

Chapter 5: Challenges Overcome ............................................................ 27

Chapter 6: Evaluation ............................................................................ 29

Chapter 7: Conclusion and Future Work ................................................. 34

References .............................................................................................. 36

Appendix A: Intortus Web API Methods .................................................. 38

Appendix B: Demo ................................................................................. 42

Appendix C: Code of Intortus Web API.................................................... 45
List of Figures

Figure 1: Overview of Wireless Sensor Network ................................................................. 7
Figure 2: Intortus and its system components ........................................................................ 8
Figure 3: Proof of concept design of solution in ................................................................... 10
Figure 4: Project scope .......................................................................................................... 16
Figure 5: Conceptual design of Intortus .............................................................................. 18
Figure 6: The eMote is Samraksh’s latest offering in its family of Wireless Sensor Network products .................................................................................................................. 19
Figure 7: Schema of Intortus database in Google app engine .............................................. 22
Figure 8: Network Architecture ............................................................................................ 24
Figure 9: Graph showing performance of various ciphers ................................................. 30
Chapter 1: Introduction

In recent years, cloud computing has emerged as a new computing paradigm to provide reliable services delivered through next generation data centers that are built on virtualization technologies, dynamically “provisioned” on demand as a personalized resource to meet a specific service-level agreement, which is established through a “negotiation”, and accessible as a composable service via “Web 2.0” technologies [7]. Consumers will be able to securely access applications and data from the cloud on demand, anywhere, and from any network device. In other words, the role of the cloud is to provide a place where application “suppliers” can make computing resources, data and applications available to clients. The applications are then hosted as “Software as a Service (SaaS)” and the cloud automatically scales to meet client demand. The Cloud provider has the responsibility to offer reliable hardware resources and necessary system software that allows a transparent hosting of services for SaaS providers. The architectural design of SaaS services usually considers new billing models that support the cloud pay-by-use policy of computation and data storage. The provision of SaaS services most of the time is given through web applications or directly through web services. Another flavor of cloud computing is “Platform as a Service (PaaS)” which is a combination of a development platform and a solution stack, delivered as a service on demand. It provides infrastructure on which software developers can build new
applications or extend existing ones without the cost and complexity of buying and managing the underlying hardware and software and provisioning hosting capabilities. PaaS can be used to extend the capabilities of applications developed as SaaS.

“Infrastructure as a Service (IaaS)” delivers computer infrastructure, typically, a platform virtualization environment as a service. This includes servers, software, data-center space and network equipment, available in a single bundle and billed as per usage.

According to the work from [9], three new hardware resource related aspects are introduced by the cloud computing paradigm:

1. The illusion and transparent use of an unlimited amount of resources. Therefore the application or service provider does not have the need to plan the amount of resources in advance; instead the cloud provider dynamically scales the resources on demand.
2. System software has to be aware that services are hosted according to a pay-by-use billing model. So the billing model is an integral part of the system software.
3. Hardware systems are designed at high scalability that is managed through virtual machine monitors according to the status of demand.

Wireless sensor networks (WSN) have been going through a revolution alongside with cloud computing with its increasing attention given their potential of enabling new solutions in various areas such as industrial, social, and regulatory areas, including industrial process monitoring and control, environment and habitat monitoring, healthcare, home automation, and traffic control. This is because WSNs couple the physical environment with the digital world. The tiny sensor nodes that comprise a WSN, are low cost, have low power needs, are easily deployable and provide multiple
functionalities such as sensing, data acquisition and processing capabilities. WSNs are, however, limited in their processing, data storage, communication, battery capability. If resources aren't sufficient for performing sensing and complex calculations for analysis in the network, processing sensor data offline is an appropriate method to solve this issue.

The cloud provides scalable resources, such as processing power and memory, and several kinds of connectable services on a virtually unlimited hardware layer [9]. Since wireless sensor networks are limited in their processing power, battery life, memory and communication speed, cloud computing usually offers the opposite, which makes it attractive for long term tracking and monitoring, and for shared use of data and services in different kind of projects or analysis tasks. In addition, if we extend a traditional sensor network to the internet via the cloud, WSNs that are dispersed but networked together can collaborate to accomplish many tasks that cannot be accomplished with a few powerful sensors or computers on a simple network comprising of sensor nodes. The integration of wireless sensor networks, aggregated with sensor data but with limited processing power, with a resource-rich cloud computing infrastructure makes the integration beneficial in terms of i) scalability of data storage, ii) scalability of processing power for data mining and analysis, iii) access to the processing and storage infrastructure anywhere in the world, iv) resource optimization, v) and using pricing as one more criteria for the IT infrastructure. An example of a scenario that can use cloud – WSN integration is a system that monitors the temperature of a server room remotely. The system can use sensors to record the temperature and use a cloud service to provide remote access. To enable such an exploration, sensor data of all types will drive a need
for an increasing capability to do analysis and mining on-the-fly. However, the computation needed for such an exploration may be more appropriately more from a cloud computing model than the sensor nodes itself. But current cloud providers do not address the issue of integrating sensor network with cloud applications and thus have no infrastructure to support this scenario.

In this thesis, we propose and develop an architecture and programming interface, Intortus, to enable this exploration of integrating sensor networks to the emerging cloud computing paradigm. Intortus lets software programmers develop and deploy applications on WSN much more quickly than traditional setup, a setup involving understanding and using the cloud provider API to access its data store.

Sensor nodes typically have limited resources, such as memory and processing capabilities, which often result in software that is tightly coupled to a specific sensor network requirement set. This tight coupling makes it difficult to reuse software components across sensor network projects, requiring costly reinvention and rediscovery of existing software solutions. Features required that makes a cloud service capable for sensor integration include message encryption in communicating between the sensor and the cloud, an interface for data storage by a developer unaware of the data structure provided by the cloud provider and secure storage of the data in the cloud. Intortus is a loosely coupled framework that provides the above capabilities so flexible enough to provide room for further capabilities.

The remainder of the thesis is organized as follows. Various application scenarios where Intortus can be used are discussed in detail in chapter 2. Chapter 3 presents
existing related work and how Intortus is a differentiator. In chapter 4, the Intortus framework is presented along with design and implementation details. Chapter 5 describes the specific challenges Intortus addressed. Chapter 6 describes the evaluation of Intortus based on the non functional requirements that is intended to meet. Chapter 7 concludes this paper with a look at future research.
Chapter 2: Cyber Infrastructure Application Scenarios

Current Wireless Sensor Networks (WSN) are deployed on land, underground, and underwater. WSN applications can be broadly classified in terms of purpose of tracking and monitoring as shown in Figure 1. Military applications such as target detection, battlefield surveillance, and counterterrorism originally motivated sensor network applications. However, advantages of sensor based military applications over traditional networks resulted in many other potential applications, ranging from infrastructure security to industrial sensing. Most of the cyber infrastructure applications using a sensor network fall under one of the categories mentioned in the figure. Some of these include applications to monitor and detect movement of a critical environment. Examples of such applications are applications for detecting human activity at a national border or detecting the reason why a burglar alarm goes off. An application for monitoring and controlling parameters of the entity being sensed is another scenario. Another example is an application that monitors and controls the temperature of a room remotely. Another cyber infrastructure application is an application for education or community centric purposes by data collection. Examples include mining behavior of an entity such as the time spent by a household watching TV or mining movement of shoppers in a shopping mall.
University Information Technology Services at Indiana University defines cyberinfrastructure to consist of computing systems, data storage systems, advanced instruments and data repositories, visualization environments, and people, all linked by high speed networks to make possible scholarly innovation and discoveries not otherwise possible. The word cyberinfrastructure is based intentionally on the word infrastructure. Most people do not care to know the details of how electricity is generated; they just want to plug their appliances into the wall and have them work. The ultimate goal for those people creating and implementing cyberinfrastructure projects is to make use of advanced
information technology systems as easy as plugging an appliance into a wall outlet [14]. Intortus is a framework that enables the creation of WSN based cyber infrastructure applications. The cyber infrastructure applications mentioned above require integration of the WSN with the cloud. In a more detailed view, the integration involves integration of hardware, communication, data storage, security and the cloud. Intortus is a framework that enables this integration for application developers creating cyber infrastructure applications easily. As seen in Figure 2, the cyber infrastructure integration of a WSN with the Internet using Intortus cloud service is being made accessible through various network devices and Web interfaces.

Figure 2: Intortus and its system components
3.1 Cloud computing solution for health care data collection

There are several related works that envision similar solutions as Intortus does. There have been various applications created involving sensor–cloud integration but are tightly coupled systems focusing on the application requirements. One such system is aCloud Computing Solution for Patient’s Data Collection in Health Care Institutions [11]. The solution involves sensor nodes attached to existing medical equipments that are interconnected to exchange services; these are integrated to the institution’s computing network infrastructure. The information becomes available in the cloud, from where it can be processed by expert systems and/or distributed to medical staff for analysis. The proof of concept design of the solution the paper mentions as seen in figure 3, is similar to the design of Intortus with a few variations. As seen in the figure, medical equipment having sensors collect data and send it to the exchange server where data is aggregated before it is sent to the cloud or storage. This is similar to Intortus wherein sensor nodes collect data and send data to a gateway. The gateway aggregates data from the Wireless Sensor Network (WSN). In [11], data from the exchange server is sent to the cloud for storage. The cloud data is then accessed by the medical staff using a content service via the exchange server again. Hence there is a bi-direction communication from the exchange server and the cloud, one for storing data to the cloud and one for accessing
data from the cloud. This approach requires the exchange server to have good processing power to communicate to and from the cloud, as well as for processing data before sending it to the content service.

![Diagram of the Intortus system](image)

**Figure 3:** Proof of concept design of solution in [11].

The gateway in Intortus uses the same hardware as the sensor node (a wall powered mote). Data from the cloud is accessed directly from a remote application like a web browser or a smart phone application using a web API with defined methods for accessing data. Hence, Intortus provides the flexibility of having a gateway with limited computation and storage capabilities compared to [11] and makes the system more loosely coupled to the gateway hardware limitations as it takes advantage of the cloud infrastructure for data storage, data processing and data access. The patient’s data
collection system design does not discuss about the security needed for health applications and is limited to medical applications.

3.2 Mobile Body Sensor Network for Health Applications

Mobile Body sensor network for health applications [2] does not use a cloud system but a central medical database of the medical organization for remote data monitoring. Just like Intortus, data in the medical database is accessed through web services. The medical database not being a cloud system, burdens the organization to maintain, update and troubleshoot the underlying server in case of any issues. It also increases the expenditure of the organization not just for maintaining the database but for providing technology to ensure scalability and performance of data retrieval and data storage.

The remote patient monitoring system incorporates security and privacy policies to protect end-to-end communication and access of sensitive medical information. One aspect of secure end to end communication in [2] is data being sent from the sensor nodes to the gateway and from the gateway to the clinical system in a secure fashion. The security requirements in end to end data communication are similar to that of Intortus with a slight variation. The remote patient monitoring system uses SSL between the clinical system, which accesses the data, and the medical database. Intortus supports both HTTPS and message encryption and authentication, using AES and HMAC (further explained in chapter) between the cloud or data storage server and the remote data access
points. Data sent from the gateway to the clinical system is sent via SSL as well. Intortus sends data from the gateway to the cloud securely too. However, data is sent securely using message authentication and encryption. Secure communication between the sensor nodes and the gateway is incorporated using message encryption and message authentication, which is similar to that of Intortus. Apart from network security, both [2] and Intortus support similar application security mechanisms such as user authentication data access policy and gateway authentication.

The architecture of the remote patient monitoring system pertains only to health applications and hence is tightly integrated to requirements of a health system. Hence, the WSN incorporated system developed cannot be generalized for other applications.

3.3 A framework for sensor cloud integration: Opportunities and Challenges

The primary goal of the framework is to facilitate connecting sensors, people and software objects to build community-centric sensing applications by integrating sensor networks to the emergent data center cloud model of computing. The work in [4] proposes a pub–sub based model to simplify the integration of sensor based community-centric applications. The pub-sub broker enables data classification by encapsulating sensor data into events. The pub-sub model provides services of event publications and subscriptions for asynchronous data exchange among the system entities. There can be more than one application subscribing for the sensor data. Intortus currently, deals with a single application and is a framework dealing with solely data storage and
communication to and fro the cloud in a secure manner. The work in [4] is what Intortus aims to provide to their customers to enable multiple applications using a collection of sensor data. To do so, Intortus aims to provide a system that enables data classification and that provides services for asynchronous data exchange amongst the various system components of Intortus. A pub/sub system can be incorporated into Intortus for collaboration amongst various applications using sensor data. The paper in [4], however, does not discuss the security requirements for such a framework and the security system of Intortus can be incorporated into the pub/sub framework for multiple applications and multiple clouds collaborating for various next generation community-centric sensing applications.

3.4 TinyDB

TinyDB is a query-processing system for extracting information from a network of TinyOS sensor devices. Solutions using TinyOS generally force the user to write embedded C code (NesC) to extract sensor data. However, TinyDB relieves the user from this complexity by providing an easy, SQL-like interface for extracting the data of interest from sensor nodes with limited power and hardware resources. TinyDB collects that data from motes in the environment, filters it, aggregates it together, and routes it out to a PC. The goal of both Intortus and TinyDB is to make the life as a programmer significantly easier, and allow data-driven applications to be developed and deployed much more quickly than what is currently possible. TinyDB and Intortus free the programmer from the burden of writing low-level code for sensor devices by providing a
high level interface. Although TinyDB is more database specific and does not deal with the cloud, Intortus can work with TinyDB extending it to the cloud [1] to form a cloud based query processing system. TinyDB can be incorporated in the WSN – gateway (a designated node that interacts with the cloud) communication as well for easy data access and aggregation at the gateway, before data is sent to the cloud.

3.5 Secured WSN-integrated Cloud Computing for u-Life Care (SC3)

SC3 monitors human health, activities, and shares information among doctors, care-givers, clinics, and pharmacies in the cloud, so that users can have better care with low cost. SC3 describes a WSN – cloud integration for health applications wherein data is sent to the cloud via a gateway. The gateway classifies the data if it is either health data or can be used to detect human activities. The gateway classifies data into health data, gyroscope and accelerometer data and imaging data, and store in a local database. There is an ontology engine which further classifies data to represent and infer higher level human activities. SC3 incorporates a basic ontology engine for data classification based on the sensor type. SC3 can be considered a basis for Intortus to develop an ontology engine at the gateway or the cloud.

Data is queried from an application to the cloud and to access the data from the cloud database, users need to authenticate themselves making the system secure. An image-based authentication and activity-based access control are proposed to enhance security and flexibility of user’s access. The idea proposed is a more complex and robust
version of user authentication compared to a simple user name and password based
authentication system Intortus incorporates. U-Life Care does not, however, consider
secure communication of data to and from cloud and gateway as well as cloud to the
remote application.
Chapter 4: Architecture and Programming Interface

Cloud data centers often involve three parties: the center, the application provider and the application user. The application providers want to be sure that the user never ‘sees’ the data center, only the application service. The business model here is closer to optimizing user experience per megawatt hour while also making the application development experience as simple as possible [4]. The direct stakeholders of Intortus here are the application providers which use the cloud services of Intortus to extend a Wireless Sensor Network (WSN) application to the Internet.

The focus of this project is to enable secure integration of a wireless sensor network to the cloud as shown in Figure 4.

Figure 4: Project scope
A designated node of the WSN called the gateway is used to connect a WSN to the cloud. The Intortus system consists of a cloud service that will be used to add data from the gateway to the cloud data store. The Intortus cloud service is used to access the data stored in the cloud.

The functions of the Intortus system are:

1. Store data from the WSNs securely.
2. Make this data available to users for viewing through web pages, to smart phone applications which will connect to Intortus and to other programs which will connect directly to Intortus.
3. Get input from users via web pages, smart phone apps or other programs to control the applications in the WSNs.
4. Allow applications to be loaded from desktop computers through the cloud service to the WSN gateways, which will then push them onto individual motes in the WSN.
As of now, Intortus provides the above first two services and the remaining are intended services which fall under future work. The main components of Intortus are the wireless sensor network, which further comprises of sensor nodes and the gateway, the cloud and clients which access the cloud giving focus to security involved in WSN – cloud integration. The clients developed so far are a web portal and a web service through which various types of web applications can be developed. The eMotes are loaded with software to collect, encode, and transmit data through wireless communication channels to the gateway. The gateway is responsible to receive data from sensors and to dispatch it to be appropriate storage service hosted on cloud. The gateway structures data in the form of an XML and sends it over to the cloud. The data is sent to the cloud using web API method calls. The web API is created from a web service which is hosted in the cloud for interoperability from .NET Micro framework at the gateway to Java in the cloud. The
web service receives requests from web applications to retrieve data from the cloud data store.

4.1 Sensor Node and Gateway

Figure 3 depicts the proposed architecture. The wireless sensor network consists of sensor nodes, also known as motes that are capable of performing some processing, gathering sensor data and communicating with other motes in the network. The choice of hardware for the mote is the ARM based Cortex-M3 platform. Figure 5 is the mote, named eMote which is under development. The eMote features the .Net Micro Framework v4.0 platform, ported to its Cortex M3 ARM microcontroller, thus enabling rapid, commercial grade application development using C#.

Figure 6: The eMote is Samraksh’s latest offering in its family of Wireless Sensor Network products
This relieves the user from developing applications on the mote using embedded C, hence making the wireless sensor network more usable for developing applications on the network. For .NET developers, this means that they can now create applications that run on a range of devices from targeted embedded devices (wireless sensors etc) to PCs and Servers and on to the cloud, all with the same programming model and tools. For current embedded developers, this means that they can now leverage the productivity gains that have been achieved on the desktop and reduce their time to market. The gateway has the same hardware as the eMote but with a difference that it is wall powered.

4.2 The Cloud and Data Store

The cloud platform available for data storage is Google App Engine [12]. Google App Engine provides on-demand compute and storage capabilities to host, scale, and manages Web applications and services on the Internet hosted in a datacenter. Applications for the host environment, which are typically web applications, can be developed with Python or Java. The language of development chosen is Java. The preferred communication method from the gateway to the cloud computing environment is based on web services, which is a proven technology and supports interoperability (between Java and .NET Micro Framework). The use of web services brings a lot of flexibility for the development of those clients, which are hosted in the customer’s environment so that the customer’s legacy systems can be easily integrated. They can be developed in an arbitrary programming language, as long web services are supported. Very important for the
development of cloud computing application is, that these applications can be developed locally, because both Google App Engine comes with a local runtime environment that also facilitates debugging before the application is deployed to the remote cloud computing environment. The data store used by Google App Engine is a Big Table structure with key-value pairs. While the Google App Engine’s data store interface has many of the same features of traditional databases, the data store’s unique characteristics imply a different way of designing and managing data to take advantage of the ability to scale automatically. The Intortus data storage API, used to store data from data aggregated at the gateway, creates a relational interface to the key value data structure so that the application providers need no knowledge of the Google App Engine data store structure while inserting data to the cloud. Data in the cloud is stored in an encrypted form for security purposes. Data can be accessed by an API which is a REST (Representational State Transfer) web service and hence can be accessed through an HTTP URL.

A data object in the App Engine data store is known as an entity. An entity has one or more properties, named values of one of several data types, including integers, floating point values, strings, dates, binary data, and more. Each entity also has a key that uniquely identifies the entity. The simplest key has a kind and an entity ID provided by the data store. The kind categorizes the entity so you can query it more easily. The entity ID can also be a string provided by the application. Data is stored in domains or of kind called ‘Applications’. Data for each application are stored as ‘Variables’. Variable here is a kind in Google App Engine terms. For an environment monitoring application,
temperature, humidity, air pressure will be example of variables of float type associated to the application or as entities of kind Variable. Here data type can be considered as a property of an entity temperature with value float. These attributes are stored in a timeline. Applications are accessed based on the user that has privileges to access the API and only those applications that belong to the user are made available to the user. Figure 6 describes the schema of how data is stored in the cloud database.

![Schema of Intortus database in Google app engine](image)

The description of the Intortus data store API methods are listed in the Appendix.

### 4.3 Security model

WSNs are being widely deployed in domains that involve sensitive information for example, healthcare and military. WSN can be exposed to malicious intrusions and attacks such as denial of service. The wireless medium facilitates eavesdropping and
packet injection to compromise the network's functioning. In addition, Intortus leaves further room for intrusion and attack due to the remote nature of data access via the cloud. All these factors make security extremely important. Furthermore, sensor nodes have limited power and processing resources, so standard security mechanisms, which are heavy in weight and resource consumption, are unsuitable. These challenges increase the need to develop comprehensive and secure solutions that achieve wider protection, while maintaining desirable network performance. Intortus concentrated on developing and integrating security in the initial phases of software design to achieve different security requirements such as confidentiality, authentication and integrity.

Several measures are taken to ensure security of the sensed, processed and stored data. Authentication, encryption and authorization measures are considered to protect the data store and processed in Google App Engine account. Security in Intortus includes message authentication and encryption, data access, user authentication and gateway authentication. The communication layer in Figure 4 addresses the security requirements of the framework.
Data encryption algorithms can be broadly classified as symmetric and asymmetric key algorithms. Symmetric key algorithms use a shared key between parties whereas asymmetric key algorithms have a public and private key. Even tough asymmetric algorithms are considered more complex than their symmetric counterparts, symmetric key algorithms, and due to their simplicity, require lesser processing power. Data Encryption Standard (DES) is an outdated symmetric key algorithm used by older systems. The DES algorithm uses a relatively short key which makes it vulnerable to cracking attempts by today’s fast computers. The Rijndael algorithm is the most secure symmetric encryption algorithm out there today. This algorithm has no known weaknesses and is the official Advanced Encryption Standard (AES) adopted by the US Government. Data encryption in Intortus is addressed by incorporating AES as the cryptographic algorithm running both at the gateway and the cloud. AES – 128 is incorporated in Intortus’ security model wherein encryption and decryption of data
happens in blocks of 128 bits. Key size of 128 bits is currently thought, by many observers, to be sufficient for the foreseeable future for symmetric algorithms of AES's quality.

Message authentication allows the sender to send a message to the receiver in such a way that if the message is modified on its way, then the receiver will almost certainly detect this. Message authentication is said to protect the integrity of a message, ensuring that each message that it receives and considers acceptable arrives in the same condition that it was sent out with no bits inserted, missing, or modified. Message authentication in Intortus is met by using Hash-based Message Authentication Code (HMAC). HMAC involves a cryptographic hash function in combination with a secret key. The secret key used here is the same key used for data encryption. The key will be burnt into the firmware of the gateway and provided as such to the user. Any change in the key will require the firmware to be updated.

One of the reasons these cryptographic standards are used over HTTPS is due to the limitation of Google App Engine not providing HTTPS for non appspot.com domains. The other reason being usage of an SSL library by embedded devices would require more processing power by the motes. The challenge addressed is avoiding writing embedded C implementation for the cryptography algorithms in the firmware at the gateway which has now been provided as a C# interface at the gateway. The gateway and the sensor nodes have Microsoft .NET micro framework ported in them. Data is stored in an encrypted form at the cloud. Data is accessed in an encrypted form as well and on accessing is decrypted at the client side or browser side of the application. This
model can be further extended to have two separate keys being exchanged, one between the gateway and the cloud and another between the remote query processor and the cloud as mentioned in Figure 3. The key exchanged between the gateway and the cloud can be used for gateway authentication as well. Intortus provides client side encryption/decryption as well as server side encryption/decryption to aid further extension and increase complexity of the security model if required.

Intortus provides a simple data access policy. Data in the cloud is stored in columns of user ID to separate each user from what data can be accessed by them. This prevents a user, unauthorized access to data that doesn’t belong to him/her. User authentication is a means of identifying the user and verifying that the user is allowed to access some restricted service. To access data on the Cloud, the user must authenticate and be granted access permissions. In case a user forgets the login credentials, the password will be sent to the registered e-mail address of the user. The user can click on a “Forgot Password” link in the admin console or by calling an API method to retrieve the forgotten password. There are two types of users according to the security policy of Intortus, who can access the cloud data. One user is the application programmer who writes an application at the gateway to send data to the cloud. This user has access to methods for sending and modifying data at the cloud. The application programmer can access data stored in the cloud by calling methods to get data from it. The other user classified is the application user who has view rights to the data store and cannot modify or add data to the cloud data store.
Chapter 5: Challenges Overcome

This section describes the challenges addressed during the development of Intortus. While some challenges are implementation-specific, we present other issues that are fundamental to the wireless sensor network and the cloud.

- Limited sensor node capabilities require low-level software optimization - Wireless sensor network nodes typically have limited processing capabilities to minimize cost and limit power consumption. Many sensor node hardware constraints directly impact the complexity of the software that can be run on the node. These limited hardware resources make it hard to implement a security platform that incorporates the security requirements of Intortus. This forced the implementation of a limited version of SSL library. Message encryption and authentication components were used from the axTLS open source library [16] and modified based on native libraries available on the eMote firmware. Apart from security, due to limited firmware capability in motes, large files such as images have to be broken down to fragments with a handshake implemented to receive the entire file.

- Sensor nodes not supporting platforms used for web development - Emotes are featured to address this issue by having .NET Micro Framework ported to the mote. Advance Encryption Standard (AES) and Hashed Message Authentication
Code (HMAC) algorithms are used by the gateway to encrypt data so that data is securely transmitted to the cloud. Since AES and HMAC are not included in the cryptography libraries of Microsoft .NET Micro Framework (but are supported in Microsoft .NET Framework), a native implementation of SSL (axTLS) for motes had to be incorporated with an interop C# interface.

- Restricting design of Intortus based on solutions provided by the cloud – A Java App Engine application's access to the classes in the Java standard library is limited to the Google App Engine's JRE (Java Runtime Engine) White list. Since the JRE does not support writing files to the cloud, large data such as images, are encoded to base 64, and stored as encrypted text in the cloud. Since Google App Engine provides development in Java or Python, the Intortus web API allows interoperability between various development platforms.

- Understanding the data store and services provided by the cloud - The Google App Engine (GAE) data store is built on top of Bigtable, a large, distributed key-value store that is built to scale. For someone used to a traditional relational database management system the GAE data store may seem incredibly restrictive. Intortus provides an API to the data store giving the user a relational database system feel to the cloud’s database. Hence the user wouldn’t need any knowledge on how to operate with the cloud provider’s data storage system.
Chapter 6: Evaluation

The evaluation metrics include: security, low cost, scalability, usability and openness.

Intortus provides low-level programming models to meet the major challenge of bridging the gap between hardware technology's raw potential and the necessary broad activities such as reconfiguration, execution, and communication. It establishes system mechanisms that interface to the various types of hardware and networks, supported only by distributed, primitive operating system abstractions. [8]

Due to limited resources and capabilities of mote firmware, handling large data is an issue. Data then is fragmented into packets and each packet is encrypted and aggregated at the gateway with a handshake protocol implemented.

One aspect of security in Intortus includes data encryption/decryption. Various message encryption algorithms exist with a broad classification of symmetric and asymmetric encryption/decryption. AES-128 is a symmetric algorithm and has been chosen due to its simplicity and good performance. Encrypting large volumes of data can have quite a high impact on CPU usage depending on which algorithm is used, and performance impacts can be one of the major issues preventing companies from implementing encryption. An embedded SSL library, axTLS, which has OpenSSL feature enabled, is used by the gateway to send data to the cloud in an encrypted form. As
it is seen in Figure 9, AES 128 performs well in comparison with the rest. The tests were performed with OpenSSL using a number of ciphers to encrypt 8 GB of data on a 1.0GHz Intel server. The Intortus cloud service uses the Java Cryptography Framework to incorporate AES - 128 for message encryption/decryption. It takes approximately 419853 nanoseconds, by the Google App Engine JVM to encrypt a 16 byte message using AES -128.

![Encryption Performance](image)

Figure 9: Graph showing performance of various ciphers [15]

The Intortus system is a low cost model as it uses the pay per use approach of cloud providers, an approach wherein a cloud user uses what he/she needs and the user only pays for what he/she uses. The result is that Intortus can keep costs low when the cloud usage is low. At the same time, Intortus can scale up capacity instantaneously to meet demand, without having to take time to purchase a new server, set it up, or enter into a contract with a new service organization. The pricing model of Intortus will depend
on the underlying pricing model of the cloud provider, Google App Engine. In addition, Intortus provides support for client side decryption, which means the clients that access the cloud data, can decrypt the data received from the cloud rather than the cloud decrypting data and sending the decrypted data to the client. This process reduces the processing power for decryption at the cloud hence reducing the CPU usage. Changing data of large data types like image helps in reducing the cost of storage in the cloud. Large size data such as images are encoded to a base 64 format before they are sent to the cloud. Cameras that store captured images as PGM format are converted to a JPEG format and then base 64 encoded. A 200KB sized image gets converted to around a 10KB JPEG image. This process quickens data transfer in the wireless sensor network apart from reducing data storage in the cloud.

Following is a simple scenario showing how billing at the cloud is reduced by data compression:

Assume a worse case scenario, an application that requires an image being sent every second to the cloud.

A 10KB sized image of JPEG format would mean approximately 1GB of data per day.
- 10KB * 60sec * 60min * 24hr = 1GB/day
- Cost of storage (GAE) = 0.15 USD/GB
- Monthly cost for storage = 1GB * 0.15 * 30 days = approx. $4.5 / month

If image is stored as a PGM, image size is around 200KB
- 200KB * 60sec * 60min * 24 hr = 17GB/day
- Cost of storage (GAE) = 0.15 USD/GB
- Monthly cost for storage = 17GB * 0.15 * 30 days = approx $76.5 / month

If data is sent to the cloud every 5 – 15 minutes, monthly cost to Google App Engine would be $0.15 - $0.50, irrespective of data type.
Scalability is addressed by the pay per use model of Google App Engine. Intortus can meet demands easily without having to take time to purchase new servers or enter into a contract with a service organization especially when CPU and storage demand increases. Google App Engine uses a “Big Table” data structure for its database. Big Table is a distributed storage system that can handle petabytes of data which can spread across thousands of servers. Google App Engine provides a data replication model for data stored in the cloud and hence for any downtime in the data store, the replicated data is used.

Intortus’ abstraction level defines its ease of use. In other words, to what degree does its interface relieve the user from dealing with the complex, low-level APIs of heterogeneous resources by providing an easy-to-use single entity of the whole system? [8] Large-scale sensor network applications usually involve various resource types, which increases the complexity of the code and the programming implementation needed to manage all these heterogeneous resources. Dealing with this heterogeneity alleviates the complexity of coordinating and adapting these diverse resources. [8] The Intortus system was provided to two embedded programmers to connect to the cloud. The integration did not take more than a couple of hours. The programmers were able to store data in the cloud and access the data from a web portal in the timeframe. An application developer with basic C# knowledge should be able to create a simple application that can securely connect a WSN to the GAE cloud and access data, in 2 – 4 hours. Intortus provides the users with a sample application which he/she can customize or integrate
with legacy applications. Without Intortus, the developer would take months to securely connect WSN to the cloud.

Openness is the ability to extend and modify the system easily as functional requirements change. [8] Traditional approaches encapsulating and hiding the implementation details result in a "black box" that's difficult to inspect and modify due to a tight coupled model with the hardware and heterogeneous components of the WSN. Intortus provides an abstraction and capabilities that enable further extension. Further extension can be incorporated by both a programmer using the Intortus system as well as a programmer developing the Intortus system. Examples of support provided to an Intortus programmer includes providing decryption support both at the client side as well as server side, development of features in high level programming languages such as C#/Java and not coding in embedded languages such as NesC, provision to add further methods to the Intortus API by incorporating an MVC framework that enables code reuse, using dependency injection to achieve simplification. Examples of support provided to an Intortus data store API user includes changing/removing data stored in the cloud by a programmer using the Intortus data store API, creating custom web applications that access the cloud data store using the Intortus data store API.
Chapter 7: Conclusion and Future Work

The project designed and implemented an architecture and programming interface for wireless sensor network (WSN) and cloud integration for applications that can use sensors to collect data. The system developed, provides easy and secure integration of a WSN to the cloud. The thesis describes the complexity involved in creating a cloud infrastructure for a WSN and challenges the system has addressed. The architecture and programming interface lets application programmers to develop and deploy applications on WSN much more quickly than traditional setup. The system is a prototype that can be used to implement cyber infrastructure applications using a sensor network. The system behaves as a baseline over which further features can be implemented.

The implementation of Intortus is at an early stage, focusing on data storage and secure communication to and fro the cloud. The existing cloud service can be used to create simple and secure cyber infrastructure applications using a WSN for data sampling and communication. There is a credible future in the deployment of self organizing wireless sensor data based on an ontology model to classify data in the sensor network. The model can be further extended to support multiple sensor networks collaborating with each other providing support to build community centric WSN based applications. Collaboration of sensor networks can be achieved by using a publish-subscribe broker at the cloud as described in [4]. Apart from remote data access, Intortus can be extended to
a framework that enables controlling and managing sensor nodes of the sensor network remotely from a web application. For controlling sensors, data needs to move from the cloud to the WSN. This can be incorporated by using events and callbacks in the API. Such a feature can also be used by smart phone applications to receive notifications on updates about any change in the cloud data.
References


[2] Mobile Body Sensor Networks for Health Applications Yuan Xue, Vanderbilt Posu Yan, UC Berkeley

[3] Xuan Hung Le et all, Secured WSN-integrated Cloud Computing for u-Life Care, IEEE Xplore, March 2010


[5] Building sensor networks using smartphones and web services: ramifications and development challenges, Hamilton Book Chapter


[14] Indiana University, University Information Technology Services, Knowledge base on cyberinfrastructure, [http://kb.iu.edu/data/auhf.html](http://kb.iu.edu/data/auhf.html)


Appendix A: Intortus Web API Methods

getVariables

@RequestMapping(method=POST, value="/application/{appId}"
public org.springframework.web.servlet.ModelAndView
getVariables(@PathVariable java.lang.String appId)

This method gets all the variables or name-value pairs associated with an application.

getTimeofVar

@RequestMapping(method=POST, value="/time/{appId}/{varName}"
public org.springframework.web.servlet.ModelAndView
getName(@PathVariable java.lang.String attValue, @PathVariable
java.lang.String appId)

This method gets the timestamp values of a variable. This application is identified by using the application Id or appId.

getValue

@RequestMapping(method=POST, value="/{appId}/{varName}"
public org.springframework.web.servlet.ModelAndView
getValue(@PathVariable java.lang.String attName, @PathVariable
java.lang.String appId)

This method gets the variable value of a name associated to an application. This application is identified by using the application Id or appId.

datastoreLimit

@RequestMapping(method=POST, value="/sizelimit ")
public org.springframework.web.servlet.ModelAndView
getDatastoreLimit(@PathVariable java.lang.String appId)

Every application has a size limit on the amount of data it can store. This method gets the size limit for an appId.
getDatostoreAvailSpace

@RequestMapping(method=POST, value="/availspace")
public org.springframework.web.servlet.ModelAndView
getDatostoreAvailSpace(@PathVariable java.lang.String appId)

This method returns the amount of free size left in the datastore.

getDatostoreUsedSpace

@RequestMapping(method=POST, value="/usedspace")
public org.springframework.web.servlet.ModelAndView
getDatostoreUsedSpace(@PathVariable java.lang.String appId)

This method returns the amount of used space in the datastore.

createApplication

@RequestMapping(method=POST, value="/application")
public org.springframework.web.servlet.ModelAndView
createApplication(@RequestBody java.lang.String body)
throws org.springframework.oxm.XmlMappingException,
java.io.IOException

This method creates an application. The XML passed along with the POST URL should be of the following format. <application> <appId><appId> <appName><appName> <userId><userId> ... </application>

Throws:
org.springframework.oxm.XmlMappingException
java.io.IOException

createVariable

@RequestMapping(method=POST, value="/application/variable")
public org.springframework.web.servlet.ModelAndView
createVariable(@RequestBody java.lang.String body)
throws org.springframework.oxm.XmlMappingException,
java.io.IOException

This method creates a variable in an application. The XML passed along with the POST URL should be of the following format containing the variable details, the application ID and the user ID. The variable value should be in bytes. The
element varType describes the data type of the value being passed. <application>
<appId></appId> <userID></userId> <variable> <varName> </varName>
<varValue> </varValue> <varType> </varType> </variable> </application>

Throws:
org.springframework.oxm.XmlMappingException
java.io.IOException

**setValue**

@RequestMapping(method=PUT, value="/variable/setvalue")
public org.springframework.web.servlet.ModelAndView
setValue(@RequestBody java.lang.String body) throws
org.springframework.oxm.XmlMappingException,
java.io.IOException

This method updates the value of a variable name in an application. The XML passed along with the PUT URL should be of the following format containing the variable details, the application ID and the user ID. The variable value should be in bytes. The element varType describes the data type of the value being passed. <application> <appId></appId> <userID></userId> <variable> <varName>
<varValue> </varValue> <varType> </varType> </variable> </application>

Throws:
org.springframework.oxm.XmlMappingException
java.io.IOException

**setAppName**

@RequestMapping(method=PUT, value="/application/{appId}" )
public org.springframework.web.servlet.ModelAndView
setAppName(@RequestBody java.lang.String body) throws
org.springframework.oxm.XmlMappingException,
java.io.IOException

This method updates the application name. The XML passed along with the PUT URL should be of the following format. <application> <appId></appId> <appName></appName> <userID></userId> <userId></userId> ... </application>

Throws:
org.springframework.oxm.XmlMappingException
java.io.IOException
```java
public @PathVariable java.lang.String id)
public org.springframework.web.servlet.ModelAndView
removeApplication(@PathVariable java.lang.String id)
public @PathVariable java.lang.String varName, @PathVariable
removeVariable(@PathVariable java.lang.String varName, @PathVariable
public org.springframework.web.servlet.ModelAndView
removeVariable
public org.springframework.web.servlet.ModelAndView
removeVariable
public org.springframework.web.servlet.ModelAndView
removeVariable
public org.springframework.web.servlet.ModelAndView
removeVariable
public @PathVariable java.lang.String appID
public org.springframework.web.servlet.ModelAndView
removeVariable
public @PathVariable java.lang.String varName, @PathVariable
removeVariable
public org.springframework.web.servlet.ModelAndView
removeVariable
public org.springframework.web.servlet.ModelAndView
removeVariable
public org.springframework.web.servlet.ModelAndView
removeVariable
public @PathVariable java.lang.String appId
public org.springframework.web.servlet.ModelAndView
removeVariable
public org.springframework.web.servlet.ModelAndView
removeVariable
public org.springframework.web.servlet.ModelAndView
removeVariable
public org.springframework.web.servlet.ModelAndView
removeVariable
public org.springframework.web.servlet.ModelAndView
removeVariable
public org.springframework.web.servlet.ModelAndView
removeVariable
setVarName
public org.springframework.web.servlet.ModelAndView
setName(@RequestBody java.lang.String body)
@PathVariable java.lang.String appId)
public @PathVariable java.lang.String varName, @PathVariable
public org.springframework.web.servlet.ModelAndView
removeApplication
This method updates the variable name. The XML passed along with the PUT URL should be of the following format. <application> <appId></appID> <userId></userId> <oldVarName></oldVarName> <newVarName></newVarName> </application>
This method deletes a variable by passing it's variable name and application Id it is contained in.
This method deletes an application along with theh variables associated with it.
```
Appendix B: Demo

Steps involved:

1. A spycam captures an image and sends it to the PC.
2. The PC runs a program to convert the .pgm format of the image to a .jpeg image and then sends the image to the eMote.
3. A C# program in the eMote runs the encryption algorithm with the image value and downloads it using the RS232 send function on the eMote to the PC.
4. The PC serial port listener on the laptop receives these encrypted fragments, puts them all together and sends the encrypted image to the cloud service.
5. The user can now see the recently acquired image on the web server.

Tools/Products used:

- ImageMagick – Used for image conversion
- Google’s REST client
- Camera OV7670
- Imote2 (interface board and programming board)
- eMote development board
- Laptop
- Google App Engine account and space
Steps for running the demo:

Follow the instructions on building the camera app on iMote2 here http://web.ics.purdue.edu/~paulshin/research_Imote2.html. The camera source code is in the tinyos-contrib. One can download it through the tinyos cvs source.

Have a shared folder between the tinyos VM and your PC. The VM can be run on VMWare player (version 3.0.0 onwards or version that installs VMWare Tools). This can be done by making sure the VM is connected to the Internet and installing VMWare Tools to enable sharing folders across the guest and host operating systems. Whatever the shared folder is, make sure you change the path of the folder in TestConversion (Line 74 and 75 in Program.cs of the project). The shared folder is where the image that is clicked from the camera is stored and from where the image will be picked up by the TestConversion program as well as where the converted image will be saved to. Vmware Tools will be required for having a shared folder setup.(Username/Password of VM: xubuntu/tinyos). The shared folder in the VM can be created in /mnt/hgfs/<shared folder name>. This location is where you save the clicked image.

TestConversion uses imageMagick, a command line tool or converting images from pgm to jpeg/gif. The installation guide for ImageMagick is present in their website at http://www.imagemagick.org/script/install-source.php. One can use the latest imagemagick software. When building imagemagick, make sure you use a modern
version of Visual Studio IDE (2008 and onwards) for building the config files in the
imagemagick folder.

Go to `/opt/tinyos-2.x/apps/cameraJpegTestSerialMod/java` in the TinyOS VM. Run the following command:

```bash
xubuntos@xubuntos-tinyos:/opt/tinyos-2.x/apps/cameraJpegTestSerialMod/java$ ./run
```

<USBPORTNO. Click and image and save it to `/mnt/hgfs/<Shared folder name>`. Save it as eg. `sampleimage.pgm`. Then, modify the TestConversion program.cs file at Line 74 with `sampleimage.pgm` and `sampleimage.jpeg` at Line 75. Run program.cs and wait for "Sending data To INTORTUS...." at the console. Once you see the message, go to http://intortus-samraksh.appspot.com/service/Login.html. Login to the portal with credentials NewUser/NewPasswd. Click "TestingImg" and view the image clicked.
Appendix C: Code of Intortus Web API

- Mapping.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mapping PUBLIC "-//EXOLAB/Castor Object Mapping DTD Version 1.0//EN" "http://castor.exolab.org/mapping.dtd">
<mapping>
  <!-- Mapping for Application -->
  <class name="com.samraksh.intortus.bean.Application" identity="appKey">
    <map-to xml="application" />
    <field name="appKey" type="string">
      <bind-xml name="appKey" node="element" />
    </field>
    <field name="appName" type="string">
      <bind-xml name="appName" node="element" />
    </field>
    <field name="variables" type="com.samraksh.intortus.bean.Variable" collection="arraylist">
      <bind-xml name="variable" node="element" />
    </field>
    <field name="users" type="com.samraksh.intortus.bean.User" collection="arraylist">
      <bind-xml name="user" node="element" />
    </field>
  </class>
</mapping>
```

```xml
<!-- --- Mapping for Variable ---
<field name="scope" type="string">
  <bind-xml name="scope" node="element" />
```

• DataController.java

package com.samraksh.intortus.controller;

import java.io.StringReader;
import java.io.UnsupportedEncodingException;
import java.text.SimpleDateFormat;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Calendar;
import java.util.Date;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import javax.xml.transform.Source;
import javax.xml.transform.stream.StreamSource;
import javax.xml.transform.stream.StreamSource;
import org.springframework.oxm.castor.CastorMarshaller;
import org.springframework.oxm.jaxb.Jaxb2Marshaller;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;
import com.google.appengine.api.datastore.DatastoreService;
import com.google.appengine.api.datastore.DatastoreServiceFactory;
import com.google.appengine.api.datastore.Entity;
import com.google.appengine.api.datastore.EntityNotFoundException;
import com.google.appengine.api.datastore.FetchOptions;
import com.google.appengine.api.datastore.Key;
import com.google.appengine.api.datastore.KeyFactory;
import com.google.appengine.api.datastore.Query;
import com.google.appengine.api.datastore.Query.FilterOperator;
import com.samraksh.intortus.bean.Application;
import com.samraksh.intortus.bean.User;
import com.samraksh.intortus.bean.Variable;
import com.google.appengine.api.datastore.Text;

import java.util.Properties;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.AddressException;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

import com.samraksh.intortus.controller.SSLController;

//http://datastore-intortus1.appspot.com/service/
@Controller
public class DataController {

    private CastorMarshaller castorMarshaller;

    public void setCastorMarshaller(CastorMarshaller castorMarshaller) {
        this.castorMarshaller = castorMarshaller;
    }

    //**************************************
    // Method: This method is used to find the space used up so far
    // in Google
    // * @return Long with amount of used space in Bytes
    // **************************************
    private long usedSpace(Entity userEnt) {
        DatastoreService datastore = DatastoreServiceFactory.get
            .getDatastoreService();
        Iterable<Entity> statKinds = datastore.prepare(
            new Query("__Stat_Kind__")).asIterable();
        long avgUser = 0;
        long avgApp = 0;
        long avgVar = 0, avgTime = 0, bytes, count;
        for (Entity statKind : statKinds) {
            String kind = statKind.getProperty("kind_name").toString();
            if (kind.equals("User")) {
                bytes = (Long) statKind.getProperty("bytes");
                count = (Long) statKind.getProperty("count");
                avgUser = bytes / count;
            }
        }
    }
}


if (kind.equals("Application")) {
    bytes = (Long) statKind.getProperty("bytes");
    count = (Long) statKind.getProperty("count");
    avgApp = bytes / count;
}

if (kind.equals("Variable")) {
    bytes = (Long) statKind.getProperty("bytes");
    count = (Long) statKind.getProperty("count");
    avgVar = bytes / count;
}

if (kind.equals("Time")) {
    bytes = (Long) statKind.getProperty("bytes");
    count = (Long) statKind.getProperty("count");
    avgTime = bytes / count;
}

Map<String, Object> PropMap = userEnt.getProperties();
int AppCount = PropMap.size() - 2;
avgApp = avgApp * AppCount;
int countVar = 0;
Iterator<?> entries = PropMap.entrySet().iterator();
List<Entity> varList = new ArrayList<Entity>();
while (entries.hasNext()) {
    Entry thisEntry = (Entry) entries.next();
    String key = (String) thisEntry.getKey();
    Query findVarProp = new Query("Variable");
    List<Entity> results = datastore.prepare(findVarProp).asList(FetchOptions.Builder.withDefaults());
    Iterator<Entity> it = results.iterator();
    while (it.hasNext()) {
        Entity ent = it.next();
        if (ent.getProperty("Application").equals(key) && key != "Domain" && key != "Password") {
            countVar++;
            varList.add(ent);
        }
    }
    avgVar = avgVar * countVar;
}

Iterator<Entity> it = varList.iterator();
int timeCount = 0;
while (it.hasNext()) {
    Entity ent = it.next();
    Query findTimeProp = new Query("Time");
    List<Entity> resultTime = datastore.prepare(findTimeProp).asList(FetchOptions.Builder.withDefaults());
    Iterator<Entity> iTime = resultTime.iterator();
    while (iTime.hasNext()) {
        Entity entTime = iTime.next();
        String timeId = entTime.getKey().getName().toString();
        }
int mid = timeId.lastIndexOf("App=");
String timeApp = timeId.substring(mid + 4,
timeId.length());

if (timeApp.equals(ent.getProperty("Application").toString())
&&
entTime.getProperty(ent.getKey().getName().toString()) != null) {
    timeCount++;
}
}

avgTime = avgTime * timeCount;
return (avgUser + avgApp + avgVar + avgTime);

private static final String XML_VIEW_NAME = "rest";
private static final long SIZE_LIMIT = 104857600;

/**************************************
* Method: This method is used to get all the Variables and their
latest
* values associated to an application
*
* @return XML with variable names, types and values
***************************************/

@RequestMapping(method = RequestMethod.POST, value =
"/application/{appId}/")
public ModelAndView getVariables(@PathVariable String appId,
@RequestParam String body) throws Exception {
    Source source = new StreamSource(new StringReader (body));
    User user = (User) castorMarshaller.unmarshal(source);
    DatastoreService datastore = DatastoreServiceFactory
    .getDatastoreService();
    Key userKey = KeyFactory.createKey("User",
user.getUserId());
    // appId = appId.substring(0, appId.lastIndexOf('.'));
    if (userEnt.getProperty(appId) != null
&&
userEnt.getProperty("Password").toString().equals(
    user.getPasswd())) {
        Query findAttsQuery = new Query("Variable");
        findAttsQuery.addFilter("Application",
FilterOperator.EQUAL, appId);
        List<Entity> results =
datastore.prepare(findAttsQuery).asList(
            FetchOptions.Builder.withDefaults());
        ArrayList<Variable> vars = new
ArrayList<Variable>(results.size());
        ModelAndView mav = new ModelAndView();
Iterator<Entity> it = results.iterator();
while (it.hasNext()) {
    Entity ent = it.next();
    Variable var = new Variable();
    var.setId(ent.getKey().getName());
    if (ent.getProperty("Type") == null)
        var.setType("NULL");
    else
        var.setType(ent.getProperty("Type").toString());
    if (ent.getProperty("Value") == null)
        var.setValue("NULL");
    else
        var.setValue(ent.getProperty("Value").toString());
    vars.add(var);
}
mav.addObject("variable", vars);
mav.setViewName(XML_VIEW_NAME);
return mav;
} else
    throw new Exception("User credentials are invalid or user
does not belong to application.");
}

/**************************************
* Method: This method is used to get the Value of a variable
* @return XML with variable name, type and value
**************************************/

@RequestMapping(method = RequestMethod.POST, value = "/{appId}/{varName}"
)
public ModelAndView getValue(@PathVariable java.lang.String varName,
@PathVariable java.lang.String appId, @RequestBody String body)
    throws Exception {
    String bodyContent = SSLController.decrypt(body);
    Source source = new StreamSource(new StringReader(bodyContent));
    User user = (User) castorMarshaller.unmarshal(source);
    Key userKey = KeyFactory.createKey("User",
    user.getUserID());
    DatastoreService datastore = DatastoreServiceFactory
    .getDatastoreService();
    Entity userEnt = datastore.get(userKey);
    if (userEnt.getProperty(appId) != null
        &&
        userEnt.getProperty("Password").toString().equals(
            user.getPasswd())) {
Query findAttsQuery = new Query("Variable");
// varName = varName.substring(0, varName.lastIndexOf('.'));
findAttsQuery.addFilter("Application", FilterOperator.EQUAL, appId);
List<Entity> results = datastore.prepare(findAttsQuery).asList(
    FetchOptions.Builder.withDefaults());
Iterator<Entity> it = results.iterator();
Entity ent = null;
int flag = 0;
Variable var = new Variable();
while (it.hasNext() && flag == 0) {
    ent = it.next();
    if (ent.getKey().getName().equals(varName)) {
        flag = 1;
        var.setId(ent.getKey().getName());
        if (ent.getProperty("Type") == null)
            var.setType("NULL");
        else
            var.setType(ent.getProperty("Type").toString());
    }
    else
        var.setType(ent.getProperty("Type").toString());
}
if (flag == 0) {
    throw new Exception("Variable not found!");
}
return new ModelAndView(XML_VIEW_NAME, "variable", var);
} else
    throw new Exception("User credentials are invalid or user does not belong to application.");
}

/*****************************/
* Method: This method is used to get the timestamp values of a variable *
* @return XML with variable name, type and value by Time *
/*****************************/

@RequestMapping(method = RequestMethod.POST, value = "/time/{appId}/{varName}")
public ModelAndView getTimeofVar(@PathVariable java.lang.String varName,
@PathVariable java.lang.String appId, @RequestBody String body) throws Exception {
    String bodyContent = SSLController.decrypt(body);
    Source source = new StreamSource(new StringReader(bodyContent));
    User user = (User) castorMarshaller.unmarshal(source);
    Key userKey = KeyFactory.createKey("User", user.getUserId());
    DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
    Entity userEnt = datastore.get(userKey);
    if (userEnt.getProperty(appId) != null && userEnt.getProperty("Password").toString().equals(user.getPasswd())) {
        Query findAttsQuery = new Query("Variable");
        varName = varName.substring(0, varName.lastIndexOf('.'));
        findAttsQuery.addFilter("Application", FilterOperator.EQUAL, appId);
        List<Entity> results = datastore.prepare(findAttsQuery).asList(FetchOptions.Builder.withDefaults());
        Iterator<Entity> it = results.iterator();
        Entity ent = null;
        int flag = 0;
        Variable var = new Variable();
        while (it.hasNext() && flag == 0) {
            ent = it.next();
            if (ent.getKey().getName().equals(varName)) {
                flag = 1;
                var.setId(ent.getKey().getName());
                if (ent.getProperty("Type") == null) var.setType("NULL");
                else var.setType(ent.getProperty("Type").toString());
                if (ent.getProperty("Value") == null) var.setValue("NULL");
                else var.setValue(ent.getProperty("Value").toString());
            }
        }
        if (flag == 0) { throw new Exception("Variable not found!"); }
        return new ModelAndView(XML_VIEW_NAME, "variable", var);
    } else
        throw new Exception("Invalid parameters!");
}
"User credentials are invalid or user does not belong to application."
}

/******* Method: This method returns the Data store Limit size *******
* @return Long value in Bytes
*****************************/

@RequestMapping(method = RequestMethod.POST, value = "/sizelimit")
public ModelAndView getDSLimitBytes(@RequestBody String body)
throws Exception {
    Source source = new StreamSource(new StringReader(body));
    User user = (User) castorMarshaller.unmarshal(source);
    Key userKey = KeyFactory.createKey("User",
        user.getUserID());
    DatastoreService datastore = DatastoreServiceFactory
        .getDatastoreService();
    Entity userEnt = datastore.get(userKey);
    if (userEnt.getProperty("Password").equals(user.getPasswd())) {
        return new ModelAndView(XML_VIEW_NAME, ",", SIZE_LIMIT);
    } else
        throw new Exception("Invalid user credentials");
}

/******* Method: This method returns the Data store Available space *******
* @return Long value in Bytes
*****************************/

@RequestMapping(method = RequestMethod.POST, value = "/availspace")
public ModelAndView getDSAvailSpaceBytes(@RequestBody String body)
throws Exception {
    DatastoreService datastore = DatastoreServiceFactory
        .getDatastoreService();
    Source source = new StreamSource(new StringReader(body));
    User user = (User) castorMarshaller.unmarshal(source);
    Key userKey = KeyFactory.createKey("User",
        user.getUserID());
    // StringBuffer msg = new StringBuffer();
    Entity userEnt = datastore.get(userKey);
    if (userEnt.getProperty("Password").equals(user.getPasswd())) {
        long usedSpace = usedSpace(userEnt);
        return new ModelAndView(XML_VIEW_NAME, ",", SIZE_LIMIT - usedSpace);
    } else
        throw new Exception("Invalid user credentials");
}
throw new Exception("User credentials are invalid");
}

/*****************************/
* Method: This method returns the Used space in the datastore
pertaining to
   * a user
   *
   * @return Long value in Bytes
*****************************/
@RequestMapping(method = RequestMethod.POST, value = "/usedspace")
public ModelAndView getDSUsedSpaceBytes(@RequestBody String body) throws Exception {
    DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
    Source source = new StreamSource(new StringReader(body));
    User user = (User) castorMarshaller.unmarshal(source);
    Key userKey = KeyFactory.createKey("User", user.getUserId());

    // StringBuffer msg = new StringBuffer();
    Entity userEnt = datastore.get(userKey);
    if (userEnt.getProperty("Password").equals(user.getPasswd())) {
        long usedSpace = usedSpace(userEnt);
        return new ModelAndView(XML_VIEW_NAME, "", usedSpace);
    } else
        throw new Exception("User credentials are invalid");
}

/*****************************/
* Method: This method creates a namespace for an Application
   *
   * @return XML with Application details created
*****************************/
@RequestMapping(method = RequestMethod.POST, value = "/application")
public ModelAndView createApplication(@RequestBody String body) throws Exception {
    Source source = new StreamSource(new StringReader(body));
    Application app = (Application) castorMarshaller.unmarshal(source);
    if (app.getAppKey() == null) {
        throw new Exception("Application key cannot be null");
    }
    Key appKey = KeyFactory.createKey("Application", app.getAppKey());
    Entity appEntity = new Entity(appKey);
    DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
if (SSLController.decrypt(app.getAppName()) == null) {
    throw new Exception("Application name cannot be null");
} else {
    appEntity.setProperty("AppName", app.getAppName());
    Key userKey = KeyFactory.createKey("User", app.getUsers().get(0)
        .getUserId());
    Entity userEntity = datastore.get(userKey);
    long usedSpace = usedSpace(userEntity);
    if (usedSpace < SIZE_LIMIT - 500) {
        if (userEntity.getProperty("Password").toString().equals(
            app.getUsers().get(0).getPasswd())) {
            userEntity.setProperty(app.getAppKey(),
                app.getAppName());
            appEntity.setProperty("Domain", userEntity
                .getProperty("Domain").toString());
            datastore.put(userEntity);
            datastore.put(appEntity);
            return new ModelAndView(XML_VIEW_NAME,
                "variable", app);
        } else {
            throw new Exception("Wrong user credentials");
        }
    } else {
        throw new Exception("Insufficient space");
    }
}

/******************************************************************************
 * Method: This method creates a variable in an application with a variable
 * name, type and value
 * @return XML with variable details created
 ******************************************************************************/

@RequestMapping(method = RequestMethod.POST, value = "/application/variable"
)
public ModelAndView createVariable(@RequestBody String body) throws Exception {
    Source source = new StreamSource(new StringReader(body));
    Application app = (Application) castorMarshaller.unmarshal(source);
    DatastoreService datastore = DatastoreServiceFactory
        .getDatastoreService();
    List<Variable> var = app.getVariables();
    if (SSLController.decrypt(var.get(0).getidd()) == null) {
        throw new Exception("Variable ID cannot be null");
    }
    Key varKey = KeyFactory.createKey("Variable",
        var.get(0).getidd());
    Entity varEntity = new Entity(varKey);
Key userKey = KeyFactory.createKey("User", 
app.getUsers().get(0).getUserId());

Entity userEnt = datastore.get(userKey);
long usedSpace = usedSpace(userEnt);
if (usedSpace < SIZE_LIMIT - 500) {
    if (userEnt.getProperty(app.getAppKey()) != null &&
        userEnt.getProperty("Password").toString().equals(
            app.getUsers().get(0).getPasswd())) {
        try {
            Key appKey = 
                KeyFactory.createKey("Application", app.
                    .getAppKey());
            Entity appl = datastore.get(appKey);
            Text textVal = new
                Text(var.get(0).getValue());
            varEntity.setProperty("Value", textVal);
            varEntity.setProperty("Type", 
                var.get(0).getType());
            appl.getKey()
                .getName());
            datastore.put(varEntity);
            if (var.get(0).getTime() != null) {
                String timeString = 
                    SSLController.decrypt(var.get(0).
                        .getTime());
                DateFormat formatter = new
                    SimpleDateFormat("yyyyMMddHHmmssSSS");
                formatter.setLenient(false);
                Date date = (Date)formatter.parse(timeString);
                //
                var.get(0).setTime(formatter.format(date));
                Key timeKey = 
                    KeyFactory.createKey("Time", var.get(0).
                        .getTime() + "+App=" + 
                        app.getAppKey());
                Entity timeEntity;
                try {
                    timeEntity = 
                        datastore.get(timeKey);
                    timeEntity.setProperty(var.get(0).getId(), textVal);
                    datastore.put(timeEntity);
                } catch (EntityNotFoundException e) {
                    Entity timeEnt = new
                Entity(timeKey);
        }
    }
    app.getUsers().get(0).getPasswd())
}
timeEnt.setProperty(var.get(0).getId(), textVal);
        datastore.put(timeEnt);
    }
    return new ModelAndView(XML_VIEW_NAME, "variable", app);
}
}
return null;
}
else
    throw new Exception("User credentials are invalid or user does not belong to application");
} else
    throw new Exception("Insufficient size");
}

// For camera demo purpose only
public static String now(String dateFormat) {
    Calendar cal = Calendar.getInstance();
    SimpleDateFormat sdf = new SimpleDateFormat(dateFormat);
    return sdf.format(cal.getTime());
}

/**************************************
* Method: This method sets the value of a variable at an instance of time
* @return XML with variable value, time it is set, and datatype
**************************************/

@RequestMapping(method = RequestMethod.POST, value = "/variable/setvalue")
public ModelAndView setValue(@RequestBody String body) throws Exception {
    Source source = new StreamSource(new StringReader(body));
    Application app = (Application) castorMarshaller.unmarshal(source);
    DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
    // Using GAE system time in UTC, subtracting 4 hours
    String time = DataController.now("yyyyMMdd");
    int hr = Integer.parseInt(DataController.now("hh")) - 4;
    String min = DataController.now("mmssS");
    time = SSLController.encrypt(time + "0" + hr + min);
    Key userKey = KeyFactory.createKey("User", app.getUsers().get(0).getUserId());
    Entity userEnt = datastore.get(userKey);
    if (userEnt.getProperty(app.getAppKey()) != null)
userEnt.getProperty("Password").toString().equals(
    app getUsers().get(0).getPasswd())
} 
    Key varKey = KeyFactory.createKey("Variable",
    app.getVariables().
        .get(0).getId().toString());
    Entity varEnt = datastore.get(varKey);
    Variable var = app.getVariables().get(0);
    if (varEnt.getProperty("Application").equals(app.getAppKey())) {
        // Text textVal = new
        // Text(SSLController.decrypt(var.getValue()));
        // String decryptedText =
        SSLController.decrypt(var.getValue());
        // int firstOcc = decryptedText.indexOf("*");
        Text textVal = new Text(var.getValue());
        varEnt.setProperty("Value", textVal);
        // Text textVal = new
        Text(decryptedText.substring(0,firstOcc));
        // varEnt.setProperty("Value",textVal);
        // Text textVal = new Text(var.getValue());
        // varEnt.setProperty("Value", textVal);
        datastore.put(varEnt);
        if (var.getTime().toString() != null) {
            Key timeKey =
                KeyFactory.createKey("Time", var.getTime().toString() + "+App=
            //        + app.getAppKey());
            // Key timeKey =
            KeyFactory.createKey("Time", var.getTime().toString() + "+App="
            //        + app.getAppKey());
            //
            // app.getAppKey());
            Entity timeEntity;
            try {
                timeEntity =
                    datastore.get(timeKey);
                    //
                    SSLController.decrypt(var.getValue().toString());
                    // timeEntity.setProperty(var.getId(),
                    SSLController.decrypt(var.getValue().toString());
                    timeEntity.setProperty(var.getId(),
                    textVal);
                    datastore.put(timeEntity);
            } catch (EntityNotFoundException e) {
                Entity timeEnt = new
                    Entity(timeKey);
                    // timeEnt.setProperty(var.getId(),
                    SSLController.decrypt(var.getValue().toString());
                    timeEnt.setProperty(var.getId(),
                    textVal);
                    datastore.put(timeEnt);
                }
            }
        }
    }
}
} else
    throw new Exception("Variable does not exist in application");
    return new ModelAndView(XML_VIEW_NAME, "variable", var);
} else
    throw new Exception("User credentials are invalid or user
does not belong to application");

/*******************
* Method: This method is used to change/set the application name
* @return XML with application details
*******************/

@RequestMapping(method = RequestMethod.PUT, value = "/application/{appId}")
public ModelAndView setAppName(@RequestBody String body,
    @PathVariable String appId) throws Exception {
    Source source = new StreamSource(new StringReader(body));
    DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
    // appId = appId.substring(0, appId.lastIndexOf(" "));
    Application app = (Application)
castorMarshaller.unmarshal(source);
    Key userKey = KeyFactory.createKey("User",
        app.getUsers().get(0)
            .getUserId());
    Entity userEnt = datastore.get(userKey);
    if (app.getAppKey().equals(appId) || app.getAppKey() ==
null) {
        if (userEnt.getProperty(appId) != null
            && userEnt.getProperty("Password").toString().equals(
app.getUsers().get(0).getPasswd())) {
            Key appKey =
    KeyFactory.createKey("Application", app.id);
            Entity appEntity = datastore.get(appKey);
            if (app.getAppName() == null) {
                throw new Exception("Application name
cannot be null");
            }
            appEntity.setProperty("AppName",
app.getAppName());
            datastore.put(appEntity);
            return new ModelAndView(XML_VIEW_NAME,
"variable", app);
        } else
    throw new Exception("User credentials are invalid or
user does not belong to application.");

59
} else
    throw new Exception(
        "Application Id in URL does not match
    with XML.");
}

/******************************************************************************
 * Method: This method deletes a variable from an application
 * @return XML with variable details deleted
 ******************************************************************************/

@RequestMapping(method = RequestMethod.PUT, value =
    "/delete/application/variable/{appId}/{varName}")
public ModelAndView removeVariable(@PathVariable String appId,
    @PathVariable String varName, @RequestBody String
    body)
 throws Exception {
    DatastoreService datastore = DatastoreServiceFactory
    .getDatastoreService();
    Source source = new StreamSource(new StringReader(body));
    User user = (User) castorMarshaller.unmarshal(source);
    Key userKey = KeyFactory.createKey("User",
    user.getUserId());
    Entity userEnt = datastore.get(userKey);
    varName = varName.substring(0, varName.lastIndexOf("."));
    if (userEnt.getProperty(appId) != null
        userEnt.getProperty("Password")
            .toString().equals(
        user.getPasswd())) {
        Query findVarProp = new Query("Variable");
        List<Entity> results =
            datastore.prepare(findVarProp).asList(
                FetchOptions.Builder.withDefaults());
        Iterator<Entity> it = results.iterator();
        while (it.hasNext()) {
            Entity ent = it.next();
            if (ent.getProperty("Application")
                .toString().equals(appId)) {
                Key varKey =
                    KeyFactory.createKey("Variable", ent.getKey()
                        .getName().toString());
                datastore.delete(varKey);
            }
        }
    }
    Variable var = new Variable();
    var.setId(varName);
    Query findTimeProp = new Query("Time");
    List<Entity> results1 =
        datastore.prepare(findTimeProp).asList(
            FetchOptions.Builder.withDefaults());
    Iterator<Entity> it1 = results1.iterator();
    while (it1.hasNext()) {
        Entity ent = it1.next();
    }
}

60
String timeId = ent.getKey().getName().toString();
int mid = timeId.lastIndexOf("App=");
String timeApp = timeId.substring(mid + 4, timeId.length());
if (appId.equals(timeApp)) {
    ent.removeProperty(varName);
    datastore.put(ent);
    Map<String, Object> PropMap = ent.getProperties();
    if (PropMap.size() == 0) {
        datastore.delete(ent.getKey());
    }
}
return new ModelAndView(XML_VIEW_NAME, "variable", var);
} else 
    throw new Exception("User credentials are invalid or user does not belong to application.");
}

/*******************
* Method: This method deletes an application and all the variables pertaining to it.
* @return XML with application details deleted
 *******************/

@RequestMapping(method = RequestMethod.PUT, value = "/del/application/{appId}")
public ModelAndView removeApplication(@PathVariable String appId,
    @RequestBody String body) throws Exception {
    DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
    Source source = new StreamSource(new StringReader(body));
    User user = (User) castorMarshaller.unmarshal(source);
    Key userKey = KeyFactory.createKey("User", user.getUserId());
    Entity userEnt = datastore.get(userKey);
    // appId = appId.substring(0, appId.lastIndexOf("."))) ;
    if (userEnt.getProperty(appId) != null &&
        userEnt.getProperty("Password").toString().equals(
            user.getPasswd())) {
        // appId = appId.substring(0,
        appId.lastIndexOf("."))) ;
        Key appKey = KeyFactory.createKey("Application", appId);
        Entity appEntity = datastore.get(appKey);
        Application app = new Application();

app.setAppKey(appEntity.getKey().toString());
datastore.delete(appKey);
Query findVarProp = new Query("Variable");
List<Entity> results =
datastore.prepare(findVarProp).asList(
    FetchOptions.Builder.withDefaults());
Iterator<Entity> it = results.iterator();
while (it.hasNext()) {
    Entity ent = it.next();
    if (ent.getProperty("Application").toString().equals(appId)) {
        String varName =
        ent.getKey().getName().toString();
        Key varKey =
        KeyFactory.createKey("Variable", varName);
        datastore.delete(varKey);
        Query findTimeProp = new Query("Time");
        List<Entity> results1 =
datastore.prepare(findTimeProp)
        .asList(FetchOptions.Builder.withDefaults());
        Iterator<Entity> it1 =
        results1.iterator();
        while (it1.hasNext()) {
            Entity ent1 = it1.next();
            String timeId =
            ent1.getKey().getName().toString();
            int mid =
            timeId.lastIndexOf("App=");
            String timeApp =
            timeId.substring(mid + 4, timeId
            .length());
            if (appId.equals(timeApp)) {
                ent1.setProperty(varName, varName);
                datastore.put(ent1);
                Map<String, Object> PropMap =
                ent1.getProperties();
                if (PropMap.size() == 0) {
                    datastore.delete(ent1.getKey());
                }
            }
        }
    }
}
userEnt.removeProperty(appId);
datastore.delete(userKey);
return new ModelAndView(XML_VIEW_NAME, "variable",
app);
} else
    throw new Exception("User credentials are invalid or user
does not belong to application.");
// created for testing purposes to create a user in Intortus
@RequestMapping(method = RequestMethod.POST, value = "/setUser")
public ModelAndView setUser(@RequestBody String body) throws Exception {
    DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
    Source source = new StreamSource(new StringReader(body));
    User user = (User) castorMarshaller.unmarshal(source);
    Key userKey = KeyFactory.createKey("User",
            user.getUserId().toString());
    Entity userEnt = null;
    try {
        userEnt = datastore.get(userKey);
        throw new Exception("User already exists!");
    } catch (EntityNotFoundException e1) {
        userEnt = new Entity(userKey);
        userEnt.setProperty("Password",
                user.getPasswd().toString());
        userEnt.setProperty("Domain",
                SSLController.encrypt(SSLController.decrypt(user.getUserId().toString())
                                + "Domain"));
        userEnt.setProperty("Email",
                user.getEmail().toString());
        datastore.put(userEnt);
        String msgBody = "Hi " + SSLController.decrypt(user.getUserId().toString()) + "!
You are now registered with Intortus.\n\nThank you!\nIntortus Admin";
        Properties props = new Properties();
        Session session = Session.getDefaultInstance(props, null);
        try {
            Message msg = new MimeMessage(session);
            msg.setFrom(new InternetAddress(SSLController.decrypt(user.getEmail().toString()), "IntortusAdmin"));
            msg.addRecipient(Message.RecipientType.TO, new InternetAddress("" + SSLController.decrypt(user.getUserId().toString()), SSLController.decrypt(user.getEmail().toString())"));
            msg.setSubject("Your e-mail account has been activated");
            String msgBody = "Hi " + SSLController.decrypt(user.getEmail().toString()) + "!
You are now registered with Intortus.\n\nThank you!\nIntortus Admin";
        }
    }
}
msg.setText(msgBody);
Transport.send(msg);
}
}

return new ModelAndView(XML_VIEW_NAME, "user",
user.getUserId()
.toString());

/**********************
 * Method: This method emails the password to the email id the
 user is
 * registered with
 * @return Returns the user ID
 *********************/
@RequestMapping(method = RequestMethod.POST, value =
"/forgotPasswd/{userId}"
) public ModelAndView forgotPasswd(@PathVariable String userId)
throws Exception {
DatastoreService datastore = DatastoreServiceFactory
.getDatastoreService();
Key userKey = KeyFactory.createKey("User", userId);
Entity userEnt = datastore.get(userKey);
String msgBody = "Hi " + userId + "!
Your Intortus password is "
 + userEnt.getProperty("Password").toString()
 + ".\n\nThank you!\nIntortus Admin";
Properties props = new Properties();
Session session = Session.getDefaultInstance(props, null);

try {
Message msg = new MimeMessage(session);
msg.setFrom(new
InternetAddress("thesamrakshcompany@gmail.com",
 "IntortusAdmin"));
msg.addRecipient(Message.RecipientType.TO, new
InternetAddress( "thesamrakshcompany@gmail.com", userId));
msg.setSubject("Intortus account information");
msg.setText(msgBody);
Transport.send(msg);
}
}

} catch (AddressException e) {
// ...
} catch (MessagingException e) {
// ...
} catch (UnsupportedEncodingException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}
}
} catch (MessagingException e) {
    // ...
} catch (UnsupportedEncodingException e) {
    e.printStackTrace();
}
return new ModelAndView(XML_VIEW_NAME, "user", userId);
}