Migrating Service-Oriented System to Cloud Computing: An Experience Report

Muhammad Aufeef Chauhan
School of Innovation, Design, and Engineering
Mälardalen University
721 78 Västerås, Sweden
mcn10003@student.mdh.se

Muhammad Ali Babar
IT University of Copenhagen
Rued Langgaards Vej 7
DK-2300 Copenhagen S, Denmark
malibaba@itu.dk

Abstract - Cloud computing has gained significant attention of industry and academic sectors which are interested in adopting or experimenting with this technology. An increasing number of companies are expected to migrate their systems to cloud enabled infrastructures. However, there has not been much attention paid to provide sufficient process support. Since migration projects are likely to encounter several kinds of challenges, it is important to identify and share the process and logistical requirements of migration projects in order to build a body of knowledge of appropriate process, methods, and tools. This paper purports to contribute to the growing knowledge of how to migrate existing systems to cloud computing by reporting our effort aimed at migrating an Open Source Software (OSS) framework, Hackystat, to cloud computing. We report the main steps followed, the process and technical challenges faced, and some of the strategies that helped us to address those challenges. We expect the reported experiences can provide readers with useful insights into the process and technical aspects that should be considered when migrating existing software systems to cloud computing infrastructures.

Keywords: Cloud Computing, Software Migration, Evolution, Open Source Software, Software Engineering.

I. INTRODUCTION

Cloud computing has emerged as an attractive area of research and practice over the last few years. It promises on-demand scalability (i.e., up or down) of information technology infrastructure without committing huge upfront investment [1, 17, 18]. There are generally three service models (i.e., Software as a Service, Platform as a Service, and Infrastructure as a Service) and several deployment models (e.g., public, private, and hybrid) of cloud computing. The Software as a Service (SaaS) model is based on the principles of Service Oriented Architectures (SOA). The SaaS model exposes a software system’s functionality to other applications. Such systems can be built on top of software infrastructure provided by Platform as a Service (PaaS) clouds like Google App Engine [5] and can utilize on demand storage resources and elastic computing resources offered by Infrastructure as a Service (IaaS) clouds like Amazon’s Simple Storage Service (S3) and Elastic Compute Cloud (EC2).

Several studies report the challenges [18] and benefits [19] of cloud computing, and the risks associated with adopting cloud infrastructure [20]. However, there is little literature available on process and methodological guidance on migrating existing software systems to cloud computing. We noticed a paucity of process and methodological guidance when we undertook a project aimed at modernizing an Open Source Software (OSS) framework, Hackystat [2], for leveraging the flexibility and scalability of using cloud computing paradigm. Hackystat is a service-oriented system for monitoring the software development projects by gathering, analyzing, and reporting product and process metrics.

To the best of our knowledge, there was no process and methodological support specifically developed for migrating existing systems to cloud computing. We decided to support our migration efforts by taking guidance from methodologies reported in [4, 10, 21, 22]. We also became interested in finding out how much support these approaches can provide to a migration project like ours. In this paper, we provide a detailed description of various aspects of our effort to migrate Hackystat [2] to SaaS system. During this project, we learned several lessons about the steps to be followed and the kind of support that can be provided by existing methods such as [10, 22]. We assert that this work can contribute to the efforts aimed at providing appropriate process, methods, and tools for supporting the migration to cloud computing.

Organization of the paper is as follows. Section II elaborates the project’s objectives and overview of the selected software system. Section III describes the process of requirements identification, analysis and evaluation of potential solutions. Steps that we followed to migrate the existing software to SaaS system are described in Section IV. Section V provides implementation and deployment details. Section VI outlines how to deal with the quality attributes consideration for such projects. Last two sections present conclusions and future work.

II. THE PROJECT AND SYSTEM OVERVIEW

This section provides the brief overview of our research objectives and selected software system that we chose to transform as SaaS system.

A. The project

This project was motivated by the need of migrating a product and process measurement framework to cloud computing infrastructure in order to monitor the development activities of large number of geographically
distributed software development teams. We wanted the system to be able adopt itself to the increasing or decreasing demands of computing and storage resources. Cloud computing offers the elastic computing and storage resource, hence, it was a suitable choice for our needs. However, we found that there was little guidance available to support the migration process in terms of steps to be followed and potential challenges involved. Hence, this project was initiated with two main research objectives:

- Migrate an OSS product and process measurement framework to SaaS model in order to support large number of distributed software development teams.
- Identify the steps and challenges involved in the migration process with the aim of contributing to the development of appropriate process support.

Further details about our project’s context, objective and outcome can be found in [14].

B. Overview of Hackystat

Hackystat is an open source framework for automated collection, analysis, and visualization of process and product data [2, 6]. It has been under development since 2001 and has undergone eight releases. Most significant enhancement was done in release 8 when Hackystat’s architecture was migrated from a client-server web application to an SOA using Representational State Transfer (REST) design principles [3, 16] and HTTP protocol for communication among services. Figure 1 shows the high level architectural view of Hackystat [2].

The DailyProjectData is another service that provides daily summary of the project’s process and product metrics. All the information through database is retrieved through SensorBase service. Telemetry service provides project and process data that is displayed by Hackystat user interface services. This service uses DailyProjectData service to produce process and product trends over time.

TickerTape and ProjectBrowser are client services that presents the analysis results to a user. TickerTape interacts with SensorBase and DailyProjectData services to get information about changes in a specific project, and it generates a status report for external applications like Nabaztag Rabbit [7] and Twitter [8]. ProjectBrowser provides a web interface to utilize Hackystat’s services for managing projects. The project data can either be viewed using ProjectBrowser or be posted to external applications using TickerTape. A single telemetry operation may invoke many operations on DailyProjectData service which in turn invoke required operations from SensorBase service to produce required information. In this way, different services of Hackystat act like a pipeline.

III. REQUIREMENTS IDENTIFICATION, ANALYSIS AND POTENTIAL SOLUTIONS

This goal of our effort was two folds: provide a process and product measurement software as a service (SaaS) for managing massively large scale globally distributed software development projects by leveraging cloud computing paradigm, and identify and understand the challenges and process of migrating a SOA based system to cloud computing. The SaaS form of Hackystat was intended to be deployed by IaaS from public or private clouds. Further details about the project can be found in [14]. We broke down these high level goals into more concrete requirements for making the required architectural and implementation changes. Since there was no external customer at the time of migration, the project team started identifying the detailed requirements by analyzing the characteristics of both SaaS and IaaS.

For the Hackystat SaaS, we identified two key characteristics: SaaS cloud has the capability to add or remove additional computing and storage resources on demand; and the scaling up or down is transparent so that users have a unified view of the services.

IaaS clouds provide environments to host applications to utilize computing and storage resources in cost effective way. To make use of additional computing resources, copies of each component should be deployed on multiple physical or logical nodes. However, the clients of an SaaS should have the consolidated view of the system, irrespective of how many actual instances are running. After analyzing the properties of SaaS and IaaS clouds, we identified the following main requirements.

R1: The system should be able to scale up or scale down to meet the performance requirements.

R2: System components should be able to be deployed on public or private IaaS clouds.
R3: Components of the system that are responsible for persistence handling should be able to take advantage of storage resources provided by IaaS clouds in future.

R4: End users of the system and external clients should have access to the system through same interfaces as provided by existing systems.

R5: At the end of the migration activity, a list of the migration steps should be available to support other similar migration activities and develop a process support.

For migration safety critical system as SaaS services, where stakeholders are actively engaged; systematic approaches for requirement tradeoff like presented by Yen and Tiao can also be used [9]. We analyzed the potential solutions against requirements using the guidelines provided in [10]. According to the guidelines, quality attributes, potential solutions, risks and tradeoffs are identified for each key requirement.

First requirement (R1) about scaling up or down the system according to the required performance needs characterized modularity and scalability. For scalability, different components of the system can be replicated on multiple physical or logical nodes in numbers that best suits the performance criteria. These components can only be deployed on IaaS clouds and not on PaaS because in case of PaaS, different components have to be refactored according to the API of the specific PaaS cloud provider. There was no immediate requirement for PaaS service so the older components with modified capability to have multiple deployments on nodes were selected.

The second requirement (R2) is associated with deployments on the public and private IaaS clouds. Portability and changeability are the quality attributes associated with this requirement. As components are expected to be deployed on IaaS clouds, so any IaaS that supports the technology on which Hackystat is build would be able to host them. As there is no risk associated with this requirement so there is no tradeoff as well.

The third requirement (R3) states that the components that are responsible for persistence should be able to take use of IaaS storage. Portability is the associated quality attributes. The proposed architecture solution for this requirement is to make a separate component to handle persistence. Other components will deal through this persistence handling resource. The use of IaaS storage, persistence handler component would require platform specific modification for portability in future. Modification to storage component would take care of all portability issues related to persistence and parts of the system continue to work without any modification.

The fourth requirement (R4) requires having a consistent view of the system for external client and components. This arises the need of a new component for each group of replicated services (for example, four running instances of SensorBase component) to provide a single and uniform view to other components and client services. The component should also handle the replicated live nodes of components transparent from the end user. There are two potential solutions for this wrapper component. One is to have a REST based wrapper component service that provides the same API as that of replicated service component it is responsible to handle (for example, SensorBase controller service for replicated SensorBase components). Other way is to have a simple web component that routes the requests to the target service without processing it. The second solution was adopted because its advantage to be independent from the interfaces of the clustered services. Hence, it requires no modification if the interfaces of the service in cluster changes and can also be replaced by some more sophisticated load balancing tools. Table 1 shows the relationship between different requirements with associated quality attributes and architecture decisions.

<table>
<thead>
<tr>
<th>Req Id</th>
<th>Quality Attribute</th>
<th>Architecture Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>Modularity</td>
<td>Replicated copies of components on multiple physical or virtual nodes to scale up or scale down computing capability.</td>
</tr>
<tr>
<td>R2, R3</td>
<td>Portability</td>
<td>R2: Components' deployment on different IaaS infrastructures. R3: Separating database implementation into new component.</td>
</tr>
<tr>
<td>R2</td>
<td>Changeability</td>
<td>Modifiability for future enhancements related to cloud computing without changing other parts.</td>
</tr>
<tr>
<td>R1</td>
<td>Scalability</td>
<td>Support of replicated components provide scalability features.</td>
</tr>
<tr>
<td>R4</td>
<td>Backward Compatibility</td>
<td>System API is exposed in form of REST URIs and that remains unchanged.</td>
</tr>
</tbody>
</table>

IV. MIGRATION STEPS

Having done the analysis of the architectural aspects of Hackystat and the potential changes required in order to meet the required requirements, the proposed solutions were implemented. The implementation phase was completed by following the steps below. We assert that these steps should be included as a process support for migrating software systems to cloud computing.

A. Evaluation of components for scalability

In the first step, all the components of the framework were investigated against the scalability requirements. During the analysis of the requirements against architecture, the replication of components on more than one physical or virtual cloud machines was found to be a means to achieve performance scalability. Different components of the framework are implemented as REST services, so these are stateless in nature. Stateless components do not retain the state of the clients and

406
multiple requests from the same client can be handled by different instance of the same component. Thus, the framework’s existing components meet the basic scalability criteria. However, the components also needed to be investigated in terms of their dependency on persistence related functionality. Analysis of the different components showed that the SensorBase had been the only component that was containing the database functionality. This component could not have been replicated because doing so would have the independent database with each instance and had not been able to handle requests for data that is stored on other instances.

B. Evaluation for Orchestration

Service orchestration is an important characteristic of SaaS cloud. When different components are deployed on different physical or virtual nodes to meet the scalability requirements, those separate deployments should appear as a single coherent subsystem to other components. This signifies the need for a wrapper component for each cluster of multiple instances of replicated system components. This new wrapper component was not only supposed to provide access to the interfaces of wrapped components but also to take care of passing service request to the appropriate component without exposing the information about the actual instance to the clients.

C. Identification of the components for refactoring

The components that could not meet the scalability requirements were needed to be re-factored. Our analysis showed that SensorBase was the only component that required modifications. It was decided to decompose this component into two smaller components. One of the new components contained the business logic and other was responsible for storing and retrieving data from the database. After the re-factoring, the component responsible for performing the business logic could be replicated to meet the scalability requirements. Each of the replicated instances could share the same database through the database component. The interfaces of the new database service are also based on the REST API.

D. Evaluation of the solution against the target cloud environment

After modifying a system as SaaS, it should be deployable on different kinds of IaaS environments. For this migration activity, our requirement was a private or public IaaS that can support the technologies on which Hackystat has been built. With the ability of replication and addition of orchestration layers, different components of the system can be deployed on public, private or hybrid clouds. For some cloud environments such as Amazon EC2 that have their own services for load balancing, orchestration layer was not required. But to make it a generic solution, this layer was included in our solution.

V. IMPLEMENTATION AND PILOT DEPLOYMENT

In this section, we briefly describe the implementation details of modifying Hackystat and pilot deployment on a public cloud. Our analysis of the requirements for orchestration suggested that the simple routing methods to delegate request to the respective services was more convenient because of two reasons. First reason is that it is not required to be modified as a result of the changes in the interfaces of the actual services, and secondly because it can easily be replaces by some more sophisticated load balancing tools. For this phase of modernizing Hackystat, we have implemented a simple algorithm that reads the information about the available services from a property file and distribute the client requests to the services in a simple round robin fashion. Figure 2 shows the implemented main algorithm that distributes the requests among different instances of the same component.

```java
if (contextPath == null || contextPath.length() == 0 ||
    servicePath != null && servicePath.length() > contextPath.length() ) {
    String contextIndex = servicePath.indexOf(contextPath);
    String contextAttributes = servicePath.substring(
        contextIndex + contextPath.length(), servicePath.length());
    String contextAttributesTrim = contextAttributes.trim();
    if (contextAttributesTrim.length() > 0) {
        String serverUri = routingManager.getServiceName(new
            ServicePointText());
        if (queryString != null &&
            queryString.trim().length() > 0) {
            targetUri = serverUri + contextAttributes +
                '?' + queryString;
        } else {
            targetUri = serverUri + contextAttributes;
        }
        response.sendRedirect(targetUri);
    }
}
```

Figure 2: Routing Algorithms of Controller Services

To implement database service, all the database method that are required to fetch, insert, update and delete the information from the database are exposed by REST URIs. Clients of the database service can use its operation through respective URIs. Like all other services, database service is also implemented using a Restlet framework. URIs can be mapped to the corresponding java classes through Router attached function as shown in Figure 3.
Figure 3: URI mapping to the corresponding java class

The text enclosed in curly brackets ({{}}) represents the variable whereas the text without brackets is the part of the URI pattern. Variables are fetched from the request in the corresponding class and are used to perform the corresponding business operation. Figure 4 shows how the required information is fetched from the URI variables in order to perform the corresponding business operation.

As a result of migrating Hackystat to cloud infrastructure, there has been only two major changes in the original architecture of Hackystat: i) database layer has been separated from the SensorBase component and ii) an additional layer has been introduced to provide orchestration when component are replicated on different physical or virtual machines. There has been no change in the functional aspects of the system as it works in the same way except that every request to different services is passed through controller layer. This layer has a separate service for each cluster of the SensorBase, DailyProjectData and Telemetry components.

We have deployed the modified framework on Microsoft windows server provided by Amazon EC2 cloud services. The architectural representation of the deployed system is shown in Figure 5. All the components of the framework including database service, SensorBase, DailyProjectData and Telemetry are running directly on the virtual instance as RESTful services. ProjectBrowser that provides the user interface for metrics is also deployed on the same instance. Controller services for SensorBase, Telemetry and Daily project data are deployed in Apache Tomcat server running on the same virtual windows server. The services as well as the clients interact with each other through this controller layer which is hosted in an application server.

The virtual instance of the operating system is associated with the elastic storage. If data stored in the database grows more than the current space of the elastic storage, additional storage space can be purchased to meet the additional requirements. As the size of the data increases, the processing of data performed by the system component will also increased. In order to keep the components performing in an efficient way, additional processors and main memory would be required. A result of the benefits provided by virtualization, main memory can be increased but the number of processors is dependent on the underlying hardware architecture as well.
If the virtualized environment cannot fulfill the additional requirements of the processing power, the resource hungry components can be deployed on a new instance and controller layer can redirect the requests to that instance. Amazon EC2 also provides features to manage load between multiple virtual instances. Since this kind of service may not be available in every IaaS cloud, especially private clouds, such support should be incorporated in a migrated software system in order to make it compatible to multiple platforms.

Different data input sensors, for example, sensors installed on a developer’s machine and version control repositories send information to the SensorBase through controller layer as shown in Figure 6. In case there are more than one replications of SensorBase service, then controller layer decides about the instance that will handle a request depending upon the routing algorithm. End users of the migrated Hackystat, for example project managers, can view the information through Project Browser component similarly as viewed prior to migration.

**Figure 6: Interaction of Data Input and End User Nodes**

![Diagram of Interaction of Data Input and End User Nodes](image)

**VI. CONSIDERATIONS FOR QUALITY ATTRIBUTES**

The SaaS system can be deployed on different flavors of IaaS and PaaS infrastructures. In order to build a system according to a certain set of requirements, the target cloud infrastructure, runtime environment supported by the cloud and network infrastructure should be considered. Different methodologies for service oriented architectures [10-11] and performance evaluation of cloud infrastructure [12] have suggested that quality attributes of the target infrastructure like transaction throughput rate and response time are as important as the quality attributes of the SaaS system itself. In this following paragraphs, we discuss the quality attribute considerations, which can be taken into consideration by following the SOA evaluation guidelines reported in [10] until there are specific guidelines for cloud computing infrastructure. Following paragraphs describe how the SOA guidelines [10] can be used in migration project like ours.

**Evaluation of quality attributes:** It is becoming increasingly well known that quality attributes of cloud-enabled applications are usually dependent upon the support provided by the target cloud environments [15]. Some of the important quality attributes to be considered are throughput, interoperability, data security and network infrastructure. In order to ensure interoperability between different components of a system deployed on heterogeneous cloud infrastructures, the target environments should be compatible to each other. Software components are often replicated in order to ensure scalability. Hence, the target platform should be investigated to ensure that replication mechanisms are supported by that platform; otherwise it needs to be implemented as application functionality. Many applications have constrains over the data storage locations and network infrastructures used to transmit data between different nodes of the SaaS systems. This requires the investigation of the storage infrastructure as well as secure networks support like virtual private network [14]. The mechanisms supported by target environment to ensure throughput and reliability also need to be considered. For example, target platform’s tooling support for load balancing and replication of virtual servers. The investigation of the known issues, limitations and training support to the system stakeholders for the target infrastructure is also an important factor.

**Handling of system failure:** Since different components of a system can be deployed on different types of public or private clouds, failure in any of the hosted services can results in failure of the whole application. A failure can also occur as a result of the resource failure or exhaustion, as well as failure of software components. The failures may be recoverable or permanent. To ensure the reliability and availability of a system according to the requirements, the investigation of the heterogeneous system for the respective quality attributes should be investigated. This requires the investigation of a system infrastructure, network support, database support and middleware services such as applications servers. Different types of failure in the system, system behavior in case of failure and the capability of the system to take proper use of error handling features of a platform should also be examined.

**Security:** The architectural security requirements should be taken into account as a cloud platform offers virtual hardware, software and network resources. That means other than infrastructure, data privacy and data access policies should also be analyzed. For simple security requirements, https can be used but where more secure network is required, a virtual private network can be a better option. However, all the communications between system components hosted on virtual nodes should be through encrypted messaging whenever applications need secure data transmission. Every type of information exchange between different components should be evaluated against the security requirements and also the evaluation of interoperability of different system components as well as communication with external components through secure communication channels.
System architecture should also be evaluated against viruses, SQL injection attacks, support for message filtering, and message level security. The target cloud infrastructure requires investigations for built in support of encrypted persistence. Or encrypted or fragmented storage requires to be managed at system level.

Many components deployed on different clouds may require access authentications and authorization for security requirements. The authentication mechanisms should be able to distinguish between requests from different components of the application and requests by client applications. Authentication can be managed by using passwords, Public Key Infrastructure or tokens. However, in some applications in which different components are stateless, tokens may not work. Implementation of the additional security mechanisms may result in decreased system performance, additional development effort and reduced interoperability. Different components of a system should be verified against the security requirements, the potential security mechanisms for the security requirements, strategies of secure communication exchanges between system components as well as between system and clients. Apart from evaluating the designed application level, the target cloud environments should also be investigated to study the kind of authentication mechanisms provided by virtual hosts.

**Orchestration:** It is hard to manage the availability and location of system components in SaaS systems whose underlying hardware and software services are frequently acquired and released. This situation causes the need of component registry just like service registry in SOA where different components can be registered along with the necessary information including name, address, interface description, security mechanisms and versions of different system components. Target cloud environments should also be investigated for built-in support to provide orchestration for different services of the system.

**VII. Conclusions**

After this migration activity, we have learned several important lessons which should be considered for supporting an effort to migrating SOA systems to SaaS.

**Target IaaS infrastructure should be considered during system design.** SaaS systems eventually need to be deployed on public or private clouds. In case of private IaaS, there are no serious bottlenecks as an organization controlling the infrastructure can tune the hardware or software configuration according to the requirements. However, a public IaaS can be problematic if the target cloud platform does not comply with the hardware and specially software requirements of a system. That is why it is important to analyze the target cloud platforms before designing the actual system. During our migration effort, we focused on redesigning the system for a private IaaS cloud. After the completion of the development process, we evaluated it for deployment on a public IaaS and found that even the revised version was not possible to deploy on some cloud services because of unavailability of the specific set of tools provided by those environments.

**Existing architecture evaluation techniques can be used to evaluate the SaaS architectures; however some quality attributes can only be validated by testing a system on a target platform.** We found that SOA evaluation methods like presented in [10] can be used to validate the architecture of SaaS systems. However, some quality attributes like performance and efficiency need to be evaluated after deployment of system on target infrastructure. There may be some stateful components of the systems that cannot support parallelism more than a certain threshold.

**Software systems consist of stateless components are easy to migrate to IaaS clouds.** Since stateless components do not maintain their state, it is easy for such systems to adapt accordingly to the scalability features as RESTful components can easily be replicated on different physical or virtual machines to meet the increasing performance requirements. A system that uses database only for persistence and do not store business logic in databases in form of triggers and store procedures can use cloud database with less re-factoring cost. A system that is using database to handle part of the business logic as well would require the modifications according to the new environments. Moreover, it is more convenient and cost effective to evolve an SOA based system to SaaS targeting IaaS clouds as compared to PaaS clouds for which components need to be re-factored according to the API provided by the PaaS provider.

**SaaS system should not use the technologies whose components cannot be hosted on separate clouds.** Each of the existing cloud platforms offers a specific set of tools and technologies. If a system is dependent on the software not provided by any of the public cloud providers, then such software can only be hosted on a private IaaS cloud that can be tuned accordingly.

**Business objectives should be carefully analyzed in order to elicit and validate requirements.** The functional requirements often remains the same but extra functional requirements that are derived from business objectives can trigger significant modifications both at architecture as well as at code level. If business objectives determine that the migration activity should result in portability to IaaS clouds then the extra functional requirements would be determined on the basis of the features provided by target IaaS clouds. Whereas if business objectives determine the portability to PaaS clouds, then system components would required to be modified according to the APIs provided by the target PaaS cloud.

**VIII. Future Work**

This paper reports our effort aimed at migrating a service-based software systems to SaaS model using cloud infrastructure. This paper also describes our observations about the applicability of existing architecture based approaches to support the migration to cloud computing. Through this migration project, we have also identified the
areas where software engineering researchers need to focus in order to support the migration of legacy systems to cloud computing [23]. The details of our observations of the software engineering aspects of migrating SOA system to cloud computing can be found [14, 23]. This paper is more focused on the migration steps, implementation and deployment details of our migration effort. While the functional requirements of a system to be migrated to cloud computing often remains the same, some new functional requirements can surface to satisfy business goals of migration to cloud infrastructure. These new functional requirements should be in compliance with the properties of the target cloud infrastructure.

Our migration effort was focused on meeting the basic criteria of cloud aware software in order to provide a platform for experimentation and evaluation of other important characteristics of cloud aware services.

One of the main motivations for cloud-enable systems is efficient and reliable software services. Reliability is relativity easy to achieve with REST style service oriented system because of the replication support. The desired level of performance is hard to achieve as it involves a large number of services communicating with each other through HTTP. We plan to explore the use of performance improvement technique like Computational REST [13] for Hackystat on cloud infrastructure.

Security is another important characteristic when dealing with different services of a system deployed on external clouds. Cryptography appears to be a viable solution for REST services. However, it should be explored how to achieve a desired level of security without compromising performance. In our work, we have separated the persistence related functionality from the business logic component to provide scalability. Mechanisms need to be investigated to ensure that this information exposed through database component’s REST API only give response to the components of the system and not to the third party clients. To this goal, we will implement the security mechanisms that allow access of APIs to only registered services.

ACKNOWLEDGMENT

Muhammad Aufeef Chauhan acknowledges the EURECA Project (www.mrtc.mdh.se/eureka) funded by the Erasmus Mundus External Cooperation Window (EMECW) of the European Commission; and Philip Johnson, founder of Hackystat project from University of Hawaii for his help in understanding Hackystat. We also acknowledge the implementation related contributions of Rui Wang and Nan Yang to the reported project.

REFERENCES