Carbon: Towards a Server Building Framework for SOA Platform

Paul Fremantle, Srinath Perera, Afkham Azeez, Sameera Jayasoma, Sumedha Rubasinghe, Ruwan Linton, Sanjiva Weerawarana, Sumisa Abeysinghe
WSO2 Inc. Mountain View, CA, USA
{paul, srinath, azeez, sameera, sumedha, ruwan, samisa, sanjiva}@wso2.com

ABSTRACT

SOA proposes an architecture that composes many services together in a loosely coupled manner, and those services may provide a wide spectrum of features like implementing Business Logic, supporting Service Orchestration, Service Mediation, and Eventing, etc. Each user would, typically, choose a subset of these features and build his architecture on them. Although it is conceptually possible to fit all the features into the same server, due to performance and modularity concerns, the functionalities are broken across several servers and deployed rather than being deployed as a single server. This paper presents Carbon, a component based server building framework that allows users to pick and choose different SOA concepts and build their own customized servers. Furthermore, the same framework enables those different features to share cross cutting concerns like storage, security, user interfaces, throttling, eventing etc., thus simplifying the server development process and reducing the footprint of the overall implementation. We present Carbon, the design decisions, and architecture while comparing and contrasting the proposed framework with other component based frameworks. The primary contributions of this paper are proposing a server building framework for SOA platform, taking initial steps towards defining and implementing such a framework, and sharing experiences of building and using the framework in real world settings. Furthermore, we propose a minimal kernel for SOA upon which the proposed platform can be constructed.

Categories and Subject Descriptors
C.2.4 [Computer Communication Networks]: Distributed Systems—Client/server; D.2.11 [Software Engineering]: Software Architectures—Service-oriented architecture (SOA)

General Terms
Design, Middleware, Components

Keywords
Component based Systems, SOA, OSGI

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

1. INTRODUCTION

Five years ago, a team from our organization, most of whom were long time contributors to Apache Web Services Project, initiated an effort to bridge the gap between SOA platform vision and implementations by building upon the Apache Web Services projects. The goal has been to integrate different Apache Web Services projects and to implement missing pieces of the puzzle to build an enterprise ready complete SOA solution. As the first outcome of this effort, Tungsten Server, which was later renamed as Web Services Application Server (WSAS), was released. It builds on Axis2, but provides new container functionalities like user management, several authentication and authorization models, persistent data storage, and User interfaces, etc., thus making it much user friendly.

However, the problem started with the second product, Enterprise Service Bus, which was a similar effort on top of Apache Synapse. The team quickly observed that both the WSAS and ESB had much code in common: for example, user management code, user interface handling code, etc. Initially ESB had code copied from WSAS, which quickly turned out to be a bad idea as both code bases started to evolve on their own, thus doubling the maintenance cost. For the following discussion, we define a server as a product (or a unit of distribution) which can run standalone in the clients environment. This particular problem of handling two servers could have been solved by refactoring out the code to a separate project and taking dependencies from both ESB and WSAS to the shared project. However, it was only a part of a more complicated problem. SOA platform comprised of many servers like Web Service Containers, Enterprise Service Bus (ESB), Service Registry, Workflow Engine, Security Server, Gadget Server, and Mashup Server, etc. that count to more than 10 servers within our SOA platform implementation. Supporting multiple Servers in the SOA platform just with shared projects in aforementioned manner would be hard, if not impossible. Actually, this leads us to a well known problem in system design, which is usually addressed through Component based Software Engineering (CBSE) where different functionalities are developed as independent units and composed at the design time. In other words, a framework that solves this problem should support following properties.

1. Each functionality should be in a one place enabling users to change and evolve it from the same place.
2. It should be able to add new functionality in the future.
3. Each server should be able to choose the features they
On the other hand, drawing the server boundaries by distributing functionalities across servers at the design time is rigid and inflexible. If servers are rigidly defined, that would force users to define their deployment architectures, not based on what they want, but based on what they are given. For an example, let us assume that Sam wants to design an architecture for his SOA solution, and when he chooses an SOA platform, it will offer him a choice of servers. However, often the features he needs are contained in multiple servers, and that forces him to acquire and deploy all the servers, which would collectively meet his requirements although they have many other features that he does not need. One on hand, he has to pay more for more servers, both at the purchase time as well as for maintenance. On the other hand, he would need more hardware to run more servers and unused features would complicate his deployment as well as the user experience. In contrast, we envisioned a framework that enables users to partition SOA features across servers based on their needs. That is to provide him with a basket of features from which he can build his server, instead of forcing him to choose among servers. This idea can be best explained through two pizza parlors. The first would offer some number of set pizzas, just like the servers that are defined beforehand, whereas, the second lets the user to have a choice of cheese, crust, topping, etc., just like the envisioned framework, which would provide greater flexibility to users.

Furthermore, we observed that within the SOA platform most component implementations need functionalities like security, data storage, and User Interface (UI). Therefore, recognizing those will enable us to build a minimal kernel to the SOA platform upon which other specific features and functionalities can be implemented. Such a kernel will simplify the architecture of the proposed framework while defining the services that can be used by component developers. Therefore, finding and defining such a kernel is also a goal of this paper.

The following section discusses the motivation and requirements of a server building framework and Section 3 describes the architecture and implementation in detail. The Section 4 discusses related works and the next section discusses how Carbon framework can make a difference to the user. Finally, Section 6 concludes the paper.

2. A SERVER BUILDING FRAMEWORK

Components provide a “unit of deployment, versioning, and replacement” (see Szyperski [12]). Components provide well defined interfaces, support introspection and reflection, and have a deployment archive format. They have dependencies on interfaces exposed by other components and can be designed at an arbitrary level of granularity (e.g. component can be a thread pool or a complete server). Components can be composed after their development (at the design time of applications or at the runtime) to create other composite components or to create end user applications. Also, components carry metadata within them, which declaratively defines their behavior. Lau et. al [9] discuss components in detail.

Furthermore, most of the state of the art component frameworks use Inversion of Control (IOC, a.k.a Dependency Injection) to manage dependencies. Within such a container, components can define their dependencies as metadata, and the runtime matches and notifies components whenever their dependencies are loaded or unloaded. This allows components to do their integration wiring at the runtime and also let those wirings to be changed dynamically, thus creating a loosely coupled system. Furthermore, with this model, it is straightforward for a new component to get access to its dependent components. Hence developers do not have to worry about passing around dependencies.

A component framework with IOC provides an ideal environment to build a complex system by composing independent components, which are developed and evolved independently. For an example, the two state of art in J2EE containers, JBoss server [8] and Apache Geronimo J2EE container [1], both use the aforementioned pattern. Therefore, they provide an ideal environment to build a server building framework.

Furthermore, a server building framework for SOA requires first class support for SOA concepts, and this can be done by supporting concepts like Web Services, WS-Security, WS-Reliable Messaging, and different transports like (HTTP, SMTP, etc.) directly through metadata of component descriptions.

SOA servers have many cross cutting concerns like security, logging, thread pools, clustering and caching, and often, those cross cutting concerns already have tools to support them. Therefore, the ability to implement and integrate cross cutting concerns in a reusable manner would simplify the server building process to a greater extent. It is desirable to provide those cross cutting concerns and other key functionalities as reusable components or services, so that the servers can be built using and composing those. Furthermore, it is desirable to have a minimal core composed of independent components: or in other words, we would like to have the kernel for the SOA platform as the basis for the proposed server building framework. The goal of having a kernel can be best described using vector spaces as an example. Each vector space has an orthonormal basis, which is a set of vectors that are orthogonal to each other, and which are sufficient to create any vector by composing vectors in the space from the basis. Similarly, with the kernel for the SOA platform, we need a set of independent (orthogonal) functionalities (vectors) that can be composed together to create all other functionalities.

Each server is composed of a number of functionalities (features). As an example, Web Service Container has support for Clustering, WS-Security, EJB as services, etc. and an SOA registry has support for store, get, and delete resources, support for browse and edit resources, support for content based search, etc. Let us start the discussion about key features needed by the SOA platform by asking what the core services that are needed by most function implementations and servers are?

Since services provide the heart of SOA platform, most functionalities in SOA are built on top of services. Furthermore, most functionalities need to store runtime data or configurations and to secure data or execution. Moreover, an enterprise SOA container needs UI support for most of its functionalities. It is worth noting that J2EE containers have tried to solve the same problem for J2EE domain where they have identified security, database connections, thread pools, logging, and transactions, etc., as the core container services. Based on our experiences, we have proposed 1) Service execution 2) Data Storage 3) Security 4) User Interface
Java/ OSGI/ Web Service Container

OSGI components may export services, have dependencies like transports (see [10] for details) as first class entities. For an example, with Carbon core, a user may include within any bundle a service.xml file, which is a metadata description of a Web Service, and Carbon runtime will detect the service and deploy the service into the Axis2 instance that is running inside Carbon runtime.

3. CARBON ARCHITECTURE

Figure 1 depicts the conceptual architecture of the Carbon platform. The system stands on Java, OSGI [11], and a Web Service Container. We use Apache Axis2 [10] as the Web Service Container and Equinox project [2] as the OSGI runtime. The first layer consists of the Carbon runtime and kernel services. Carbon runtime supports the integration with Axis2, provides the base runtime, and processes and supports the component metadata contained in the Carbon components. Kernel Services provide a kernel for SOA platform as we discussed in the earlier section. The next layer implements Carbon components that provide server (product) specific functionalities. Finally, a suite of servers (products) stands on top of aforementioned layers, where each product is composed of a subset of components selected from the pool of components provided by Carbon.

Let us dive into each part of the architecture in detail. OSGI (see Rellermeyer et al. [11] for details), which is the base of Carbon, is a component framework that enables users to build their applications as a set of loosely coupled components called OSGI components. Each component is packaged as a bundle, where a bundle is a jar file, which is annotated with additional metadata about its behavior. An OSGI container supports and enforces those metadata. OSGI components may export services, have dependencies to other components, and export some java packages and classes to the outside world. An OSGI service is a java interface exposed to other components in the system, and components can register themselves with the OSGI container and get notified about selected services in the environment. Hence, OSGI Container also acts as an IOC container (as described above) by allowing bundles to discover services from each other, wire themselves at runtime, and dynamically rewire themselves when components join and leave.

By choosing OSGI as the basis, Carbon has chosen a component framework with IOC support as its foundation just like JBoss and Geronimo J2EE containers. Advantages of IOC support are discussed in Section 2. The heart of the Carbon platform, Carbon core, extends OSGI and provides following functions.

1. provides a server runtime for Carbon platform by integrating Apache Axis2 and OSGI while exposing all core Axis2 services as OSGI services.

2. extends OSGI components by defining the Carbon components. A Carbon component supports Web Services, WS-* specifications, and other Axis2 concepts like transports (see [10] for details) as first class entities. For an example, with Carbon core, a user may include within any bundle a service.xml file, which is a metadata description of a Web Service, and Carbon runtime will detect the service and deploy the service into the Axis2 instance that is running inside Carbon runtime.

3. Cabron core implements kernel services for SOA platform (execution, data, security, and UI) and exposes them as OSGI services to the entire Carbon platform so that any bundle can discover them and use them.

4. Carbon core provides a UI framework, where a user can include JSP pages and metadata about how those JSP pages fits into the overall Carbon Web Console within bundles, and Carbon core renders those JSP pages appropriately within the Web Console.

Figure 2 shows the architecture of Carbon. Since Carbon is a component framework for SOA, its main function is to serve various kinds of SOA requests. Hence, Carbon framework runs within a Servlet called Bridge Servlet, which is setup in such a way that all requests to the Servlet container are redirected to that Servlet. To start a Carbon based server, a user starts an embedded tomcat server, which in turn starts the Bridge servlet. The Bridge servlet initializes the OSGI runtime, which loads Carbon components, starts Axis2 as an OSGI component, and initializes the Carbon UI framework. All requests to Carbon go through Bridge Servlet, which redirects it to axis2 if it is a Web Service request and redirects it to UI framework if it is a UI request. Both Web Services running in Axis2 and UI components use Carbon features like IOC to build a dynamic and loosely coupled implementations.

Furthermore, we have extended the bundle loading code of OSGI to inspect the bundles, to detect metadata associated with SOA artifacts, to process them, and to convert them to runtime artifacts within the Carbon runtime. For example, this code converts service descriptions to Axis2 services and incorporates the UI definitions into the UI framework.

Carbon core also includes kernel services that act as the basis for implementing other Carbon components, and the following section describes the implementation of each kernel service. Carbon exposes each kernel service as an OSGI service so that the other Carbon components can discover them through dependency injection support provided by OSGI and use them.

Service Execution: Carbon uses Axis2 to implement services, and it converts SOA artifact descriptions included in Carbon components and deploys them into Axis.
Security: Carbon core includes a User Manager that keeps track of current users and lets Carbon components to authenticate any user. Furthermore, it includes a permission storage, which helps Carbon components to authorize users. For example, if the eventing component needs to authenticate a user, it may pass the user name and password to OSGI authentication service and check authentication of the user. Users may be authenticated either through HTTPS + HTTP basic authentication or through WS-Security, and security is enforced either at Bridge servlet or at a message interceptor inside Axis2. In the same manner, a Carbon component can use the OSGI authorization service to check the access permission of a logged-in user.

Data Storage: implemented as a registry which is backed by a database. It allows users to store anything in a hierarchy through a set( .. ) operation and retrieve them back through a get( .. ) operation. Registry is closely integrated with the security and enforces the security of data stored in the registry.

User Interfaces: Carbon provides a Web console to the user that has a left hand panel with a series of menus and sub menus. Carbon components many include UI definitions expressed as JSP pages, and through bundle metadata, users may express the position in that menu where those JSPs should be placed. Carbon UI framework dynamically generates a web page by placing those user defined web content (via JSP pages) in correct places according to metadata definitions.

Miscellaneous Services: There are many other cross cutting functionalities like threadpools, logging, throttling, caching, clustering, etc. Carbon usually uses well known implementation for each (e.g. Log4J for logging, Axis2 clustering for clustering, etc.) and exposes them as OSGI services so that Carbon components can locate them and use them. On top of the Carbon runtime sits Carbon components, and each component implements a SOA platform functionality. Each Server (Product) consists of several components, where some are shared while some are specific to the server. To understand how it works, let us consider the Eventing Server product of SOA platform. The Eventing Server consists of a Web Service which implements the WS-Eventing specification, another Web Service which supports admin operations for the broker, a UI to show current subscriptions and enable users to change configurations, and a data storage for storing configurations. On this setup, the eventing broker is implemented as few bundles, which include web services and UI develops as JSPs. Once those bundles are installed, the Web Services get deployed into the Carbon server and the UI appears in the Web console menu. Similarly, to store data, the broker component should find the OSGI registry service and store data in it, and for security, the broker component can look up the security service and use that to authenticate or authorize the users. Each of the other functionalities are also implemented as Carbon components. Also, Eventing Server reuses some common components like logging, service configurations, etc.

In this setting, if any server (e.g. BAM) decided that it needs the event support, all it has to do is to drop the Carbon components for events into a folder, and both services as well as UI components get added to the BAM server. Each of the functionalities in the SOA servers (e.g. ESB, WSAS, BPS, etc.) are implemented as a collection of Carbon components. Consequently, it is very easy to create a new server by reusing components, and also Carbon gives the users an option of picking and choosing functionalities they need for their server. Also, OSGI functions with one component instance of each type. Therefore, if two servers (e.g. BAM and Event) are merged, functions that are in both the servers are not duplicated.

One of the important questions is the possibility of any potential conflicts among Carbon components when they are used in different combinations. However, from our experience, possibility of such a conflict is minimal as long as Carbon developers adhere to few best practices. For example, each component should use a separate directory within the registry to store their own configurations and any runtime data, and any communications across components must be done through either OSGI services or Web Services. On such a setup, the co-existence of multiple Carbon components look much similar to a system composed of multiple Web Services, where the chance of conflict due to different combinations of its parts are minimal. Therefore, right now, we do not do any special handling to resolve conflicts among components.

To measure the performance impact of the new approach, we have deployed the same Web Service within Carbon based WSAS and Apache Axis2, measured throughout and latency for each (on a Intel(R) Core Duo 2.4 GHz 8G machine). As shown in Table 1, carbon imposes about 10-15% overhead in both latency and throughput, and also observed a memory hit where the consumption was about 90-170MB with carbon and 20-55MB with Axis2. Memory hit is caused by the OSGI framework, which tend to use more memory, and we believe latency and throughput overheads are caused by high memory and the bridge servlet. We are exploring the performance impact of the platform further.

<table>
<thead>
<tr>
<th></th>
<th>TPS(req/sec)</th>
<th>Latency(ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallelism</td>
<td>20 40 60</td>
<td>20 40 60</td>
</tr>
<tr>
<td>WSAS (Carbon)</td>
<td>1279 1475 1543</td>
<td>16 27 38</td>
</tr>
<tr>
<td>Axis2</td>
<td>1504 1663 1700</td>
<td>13 24 35</td>
</tr>
</tbody>
</table>

4. RELATED WORK

For the best of our knowledge, Carbon is the first attempt to build a server building framework that lets users build a custom server by choosing features from SOA and the first attempt to define a kernel for the SOA platform. However, there are many related works in the areas of J2EE, SOA, and component based Software Engineering. Let us look at each of these in detail.
Among similar efforts at J2EE, Fleury et al. [8] present the architecture of JBoss Application Server, where authors argue that containers themselves should be built on top of a component based model so that they themselves can benefit from components. Using the same idea, Carbon applies component frameworks to implement the SOA platform architecture. JBoss is built on top of JMX, but extends JMX by introducing “JBoss service component” that has additional support for service life cycles, dependency management through Inversion of Control, packaging, deployment and reconfiguration. Apache Geronimo [1] also provides a much similar model where the container itself is built using loosely coupled components called GBeans that are built with Spring IOC Container [3]. Furthermore, Desertot et al. [7] describe an effort to re-architect JoAS Server based on OSGI, and they use a much similar model where each J2EE component finds services offered by JoAS Server through OSGI. All these efforts focus on J2EE while Carbon focuses on SOA. However, both use much similar concepts. One important difference is that they only build a one server, while Carbon handles details of building a server building framework.

There have been many component frameworks proposed and Lau et. al [9] provide a detailed survey of component frameworks. For example, Yasmin [6] component framework introduces hot deployable artifacts called droplets that have well defined interfaces, and they could change its behavior at runtime. However, Yasmin does not manage dependencies among components. Among more sophisticated models, fractal components [4] provide introspection, runtime reconfiguration, hierarchical composition, and its sharing facilities are much more powerful than J2EE style component frameworks. Also, it handles dependencies by wiring interfaces required from clients together with interfaces provided by services. It also has a detailed control model, which enables users to implement monitor, control, and execute loops through controllers associated with components. Although, Carbon uses OSGI, it is theoretically possible to build it on top of a different component framework that has IOC.

Apache Web Service project implements most of the SOA platform, but as independent projects, and Carbon, in facts, is a component based integration platform for those projects. Among efforts to componentize SOA platform, Collet et al. [5] bridge SOA and fractal components. For each component they automatically generate a service and for each service they provide a proxy to bridge calls from components. This allows users to bring in a service to a system built on top of components or integrate a component based system with a SOA based system. In contrast, Carbon focuses on using a component based model to build a server building framework for SOA.

5. HOW DOES IT MAKE A DIFFERENCE?

This section describes few usecases where Carbon platform could make a difference in real life settings.

First and foremost, Carbon is a component framework for SOA built upon OSGI, and it provides first class support for SOA artifacts within a component environment. For a user who would want to build a SOA based solution while using the rich component features (e.g. OSGI), Carbon platform would provide an ideal programming environment.

Carbon allows designers to flexibly design server boundaries (what features are included in the server and what are not) and to reuse features across servers at the design time, which is a feature that can be already found in few component frameworks. However, one of its key features is allowing users to do the same after development. Carbon lets users pick and choose the features, hence lets him build the server that fits his usecase. Carbon also automatically detects the required dependent features and adds them to the server. Following are few potential usecases.

1. If a user finds that a server lacks features he needs, he can choose to add them by selecting and installing associated components. One of the common usecase is adding the Business Activity Monitoring (BAM) feature to ESB to monitor the messages that flow through ESB, thus creating an ESB+. This is very useful with features that are used by a specific audience, like Financial Information eXchange (FIX) Protocol, where only required users can choose to add them.

2. Users may find that there are many features in his server that they do not need and they do not use. They can choose to remove them for better security (to follow the principal of least privilege or to reduce potential risks: e.g. remove delete in the registry if the users never need to delete resources in the usecase). Also, the same idea can be used to build a lightweight version for a restricted environment (e.g. mobile).

3. Users may want to distribute the work done by the same server across many machines, and he can use Carbon to break up one server(e.g. ESB) to many servers by distributing Carbon components across many machines, thus distributing the load. However, it is worth noting this is possible only when the features to be distributed do not have any dependencies.

Furthermore, Carbon platform brings together most cross cutting concerns through kernel services. In other words, if we take any two servers, security is applied the same way, most code is shared, and UI and user experience is the same. The same is true for all kernel services and services like caching, clustering, logging, etc. The model is architecturally simple and this allows developers to develop, verify, and evolve cross cutting concerns in the same way from a single place. On the other hand, since cross cutting concerns are well defined and independent, it is much easier to implement new functionality by developing new Carbon components. This allows component developers to focus on the key functionality of the component. In our organization, we have observed that this significantly improves the time taken to develop new components or to integrate existing tools with Carbon platform.

In the same manner, building a new server is much easier as most common functions (like logging, monitoring, and security UIs) come from existing Carbon components or the core. This lets servers focus on their core functionality and also increases the reuse.

One of the unique features of Carbon is that it also allows users to implement User Interfaces as a part for each component. Consequently, when a new feature is added, Carbon is able to extend not only its backend, but the behavior of the User Interface as well. For example, when a user adds the content based search menu has appeared in his UI. Furthermore, since both backend functionality as well as UIs can be
reused, resulting the UIs are consistent and uniform across the platform. For example, within any server, the user management user interface is the same.

Furthermore, due to its OSGI integration, it supports discovering and automatically installing new features (implemented as Carbon components) into a server. For example, a user can start with Carbon Server (this only has the kernel, nothing else) and then creates his own server by searching for features (Carbon components) and selecting the features he needs through a User interface. Then Carbon will download new features and install them in the same way it is done with eclipse software updates.

With Carbon, it is much easier to extend a server. For example, let us consider the case where we wanted to add content based search to the registry. Content based search included an interceptor that intercepts “Put” and “Delete” operations for resources, a service to expose the search, and a Web UI to support manual search. Had the Registry not been done with Carbon, this would have required changes to the registry, but with Carbon, this functionality was implemented without any change to the registry at all, and a user can add the new feature by adding new bundles to the registry.

Since the servers built on Carbon platform are built by composing components from the same set and since they all stand on top of same kernel services, UIs, APIs, startup shutdown commands, deployments and configurations, and management, etc. are consistent across the platform. This improves the user experiences, and often, users who are comfortable with one or more products can easily find their way around all other products. Furthermore, this reduces underlying concepts associated with Carbon and hence brings the learning curve down quickly for more experienced users.

Single framework that underlies the complete SOA platform and the unified behavior have enabled us to build a consistent governance solution to manage and keep track of the complete platform. As an example, it allows users to build a solution architecture where all data, users, configurations are kept in one place. Moreover, Service Discovery, Life Cycle management, and BAM, etc. can be easily enabled. For instance, if a user needs service discovery, all he has to do is add discovery Carbon components into each of the servers, and then the complete framework supports discovery.

6. CONCLUSION

SOA platform includes many features, which are packaged under several servers (e.g. ESB, Workflow Engine, Web Service Container). Carbon framework enables developers to develop the complete SOA platform as a collection of components that they can compose into different servers later. Furthermore, Carbon extends this model so that users can build their custom servers by picking and choosing components or extending existing servers by adding new components that will add new functionalities.

Carbon platform achieves these goals through three steps. First, it uses OSGI as its underline component framework. Second, it extends the OSGI by providing first class support for building SOA servers. It does so by integrating a Web Service Container (Apache Axis2) with OSGI and supporting SOA artifacts like Services, Transports, and Web Service extensions directly through metadata embedded in OSGI bundles. Third, it defines Security, Data, User Interface, Service Execution and Miscellaneous services as the kernel for the SOA platform and makes them available to other components through Inversion of Control (IOC).

Although we provided added functionality, it is worth noting that we still release products (servers) as a set of prepackaged servers, because for a new user, that is the most convenient option. Yet, for a more advanced user, Carbon provides much more functionality and flexibility as discussed.

We have been working on the Carbon platform for three years, and now in its second generation, it powers the complete SOA platform and more than 10 server implementations in our organization. We have observed 10-15% performance degradation between the earlier version of the servers and the versions based on OSGI and a memory overhead compared to old versions. On the other hand, Carbon platform has streamlined and transformed our development process, and currently, building upon Carbon platform, we have been able to design and develop new Servers (products) within a span of few months. We believe that even if there is a slight overhead, overall benefits of the platform far outweighs the cost. Complete Carbon platform and all products built on top of it are available as Open Source under apache license, and we believe that the same concepts will be applicable outside our organization as well.

7. REFERENCES