Multi-tenant SOA Middleware for Cloud Computing

Conference Paper · July 2010

DOI: 10.1109/CLOUD.2010.50 · Source: DBLP

Srinath Perera
WSO2
82 PUBLICATIONS **1,058** CITATIONS
SEE PROFILE

Prabath Siriwardana
University of Moratuwa
1 PUBLICATION **119** CITATIONS
SEE PROFILE

Sanjiva Weerawarana
Lanka Software Foundation
122 PUBLICATIONS **13,064** CITATIONS
SEE PROFILE

Paul Fremantle
WSO2
45 PUBLICATIONS **1,200** CITATIONS
SEE PROFILE

Multi-Tenant SOA Middleware for Cloud Computing

Afkham Azeez, Srinath Perera, Dimuthu Gamage, Ruwan Linton, Prabath Siriwardana,
Dimuthu Leelaratne, Sanjiva Weerawarana, Paul Fremantle
WSO2 Inc.
Mountain View, CA, USA
{*azeez, srinath, dimuthu, ruwan, prabath, dimuthul, sanjiva, paul*}*@wso2.com*

***Abstract*—Enterprise IT infrastructure incurs many costs ranging from hardware costs and software licenses/maintenance costs to the costs of monitoring, managing, and maintaining IT infrastructure. The recent advent of cloud computing offers some tangible prospects of reducing some of those costs; however, abstractions provided by cloud computing are often inadequate to provide major cost savings across the IT infrastructure life-cycle.**

Multi-tenancy, which allows a single application to emulate multiple application instances, has been proposed as a solution to this problem. By sharing one application across many tenants, multi-tenancy attempts to replace many small application instances with one or few large instances thus bringing down the overall cost of IT infrastructure. In this paper, we present an architecture for achieving multi-tenancy at the SOA level, which enables users to run their services and other SOA artifacts in a multi-tenant SOA framework as well as provides an environment to build multi-tenant applications.

We discuss architecture, design decisions, and problems encountered, together with potential solutions when applicable. Primary contributions of this paper are motivating multi-tenancy, and the design and implementation of a multi-tenant SOA platform which allows users to run their current applications in a multi-tenant environment with minimal or no modifications.

***Keywords*-multi-tenancy, SaaS, PaaS, Cloud Computing, SOA**

I. Introduction

Enterprise IT infrastructure incurs many costs ranging from hardware costs and software licenses/maintenance costs to the costs of monitoring, managing, and maintaining IT infrastructure. The recent advent of cloud computing [1] offers some tangible prospects of reducing some of those costs; however, abstractions provided by cloud computing are often inadequate to provide major cost savings across the IT infrastructure life-cycle. This is due to the gap that exists between users and cloud computing technologies where what users wish is to outsource their complete IT infrastructure as a service, whereas cloud computing only provides much lower levels of virtualization.

Software as a Service (SaaS) [2] has been proposed as a solution to this problem, where users can outsource their complete application to a third party to manage and maintain. For example, with Google Apps, many organizations have outsourced their Email, document handling, and internal web sites to Google. Similarly, Salesforce has enabled businesses to outsource critical business side applications like customer relationship management to an outside entity. The SaaS vision strives to create an environment where users can buy customized software as a utility just like you would buy electricity or water. This would be expected to bring down the cost of IT operations by applying economies of scale by consolidating operational cost of many organizations together. In a world where this vision has been realized, running your own IT operations will be a thing of the past, just like we do not run power stations ourselves.

Providing SaaS for applications such as Email has been realized by now to a great extent, but same is not true of outsourcing Service Oriented Architecture (SOA) [3] artifacts. Service Oriented Architecture applications often consist of collections of artifacts that include:

- service implementations: some containing business logic, some providing data access, some fronting legacy applications to offer a service interface and so on,
- business processes: orchestrations of service and human interaction tasks that must be completed to achieve a particular business interaction,
- Web applications: Web based applications that allow users (customers, partners, suppliers etc.) to interact with enterprise services and transact business

Most organizations today expose SOA artifacts like Web services or business processes directly to customers, partners and suppliers. Furthermore, their internal applications and external Web applications often depend heavily on SOA artifacts in the back-end infrastructure.

With the explosive growth of mobile devices, any organization that offers public services must be ready to deal with burst load at levels never seen before. For example, an organization that offers weather data and has mobile device applications may get hit with millions of requests within a very short period of time should a catastrophic event like an earthquake occur in a particular area. Unlike in the old days, today everyone has a computer (their phone) and is constantly connected (wirelessly) - thus services that they access may get bursty loads at unpredictable times.

The old solution to this was to allocate resources to deal with expected peak demand. While that was indeed a possible solution earlier, it is no longer even practical given the potentially exponential load one may get at an unpredictable moment due to external events. What is preferred is to have a shared infrastructure where many organizations can pool their workloads so that one can achieve much greater economies of scale and at the same time have much better capacity for dealing with bursty loads.

Providers of infrastructure to run a customer's (SOA and similar) artifacts are commonly referred to as Platform as a Service (PaaS) providers. PaaS is about offering a platform that customers can deploy their components that in turn either offer services to others or are used to build SaaS applications.

When a PaaS provider offers a platform to customers, they need to ensure complete isolation between the customers and provide appropriate SLA guarantees. This is critical as often even competitors may be using the same PaaS provider.

First generation PaaS providers simply offer a physical machine (or a cluster of them) with the appropriate software installed per the customer's needs. However, this does not provide an architecture for scaling to deal with burst loads nor offer any approach for sharing. The second generation of providers use operating system level virtualization to create virtual machines per customer that share a single server or a cluster of real servers. This is for example what Amazon offers with Amazon Relational Data Service - a machine running an instance of MySQL specifically for the customer. While that does serve to significantly improve data center utilization by allowing one to share hardware capacity, it still requires a fixed allocation of hardware resources (in particular memory) per virtual machine, whether or not the VM is using such resources. However it does provide so-called elasticity - or the ability to grow and shrink the number of (virtual) servers allocated to a problem based on load. Amazon's Elastic Load Balancer is a typical example of this.

While elastic allocation of virtual machines does lead to significant improvement over earlier approaches, further improvement is possible. The next level is for PaaS providers to deploy multi-tenancy: or essentially to virtualize the middleware servers that are running the SOA artifacts and allow a single instance to be securely shared between tenants or different customers. Combining virtualization, elasticity and multi-tenancy results in optimal usage of data center resources as it means CPU, memory and network resources are maximally deployed.

Motivated by the success of SaaS, multi-tenancy has wide traction, and there have been many efforts to build multi-tenant frameworks. However, as discussed in Section II in detail, most efforts have been focused on providing data multi-tenancy. However, data only solutions only solve part of the problem (e.g. [4]) as users need to run there application/services stack, which also incurs significant overhead. In this paper, we present an architecture that enables multi-tenancy at the SOA platform level, which enable users to run their services, business processes and other SOA artifacts in a multi-tenant SOA framework. This framework then becomes a platform for building multi-tenant SaaS applications as well.

This paper is organized as follows: Section II discusses related work in the field of multi-tenancy in general and SOA multi-tenancy in particular. Section III defines the goals of a multi-tenant SOA platform. Then section IV presents our SOA platform and its implementation. Section V presents some performance evaluations of the proposed platform. Section VI reviews the security aspects of the proposed platform. Finally, Section VII wraps up the paper and identifies a series of further problems that are yet to be solved.

II. Related Work

Initial work on multi-tenancy was done in the context of Application Service Providers (ASPs) [5], which was a wave that happed in late 1990s and early 2000s. The ASP concept had goals very similar to the goals of the SaaS model and motivated multi-tenancy. However, these approaches focused on running an instance per tenant, and they fall under the level 1 of maturity model proposed by Chong et al. [4], which we will discuss shortly.

Data multi-tenancy is the most explored approach under multi-tenancy, and is often implemented on top of a database. Both Jacobs et al. [6] and Chong [7] have outlined three main approaches for data management in a multi-tenant deployment: separate databases, shared database with separate schemas and shared database with shared schemas. Among these, the separate database approach gives each tenant its own database, the shared database with separate schema approach gives each tenant its own tables, and in the shared database with shared schema approach shares the same table among many tenants and enforces security at the next layer in the architecture. They make a continuum where more isolation means less sharing & less scalability and vice versa. Among these, the final approach scales out best, due to least overhead per tenant and our proposed architecture has chosen that approach for data management. However, with that approach additional effort is required to provide total data isolation. Complex data partitioning strategies need to be adopted when the volume of data grows since all the data reside in shared tables. Aulbach et. al. [8] describe such a schema mapping technique where they introduce the concept of Chunk Folding, where the logical tables are vertically partitioned into chunks and placed in different physical databases, and joined when necessary.

Guo et al. [9] discuss execution and administration isolation, and similar to the proposed approach, they have achieved it by limiting the user to the part of the directory hierarchy and using Platform specific security modes (e.g. Java security) to ensure isolation. Moreover, they use data encryption to secure stored data. They further provide performance isolation through controlled resource allocations (e.g. quota, monitoring resource usage and enforcing priorities), which prevents a tenant from being affected by load on other tenants. Our proposed solution has not yet considered the topic of performance isolation and we plan to address it in due course.

Chong et al. [4], in one of the first discussions of multi-tenant applications, have proposed a maturity model where higher numbers indicate higher level of resource sharing. For instance, level 1 provides an instance per tenant, level 2 provides a configurable instance per tenant, level 3 runs a single instance that serves all customers, and finally, level 4 enables level 3 to scale up by running multiple instances and load balancing to scale it up. As we shall discussed in the next section, the proposed approach can be placed in level 4. Our approach addresses scalability and load balancing with a load balanced elastic approach. Configurability is provided through a management portal and via server-side tenant specific customizations.

One related solution is provided by virtualization technologies, where separation is done at the hardware level. Although use of virtualization enables different virtual hosts to share a common set of resources, each host has to run its own operating system, and therefore, the solution incurs much overhead than server level multi-tenancy. Similar arguments can be made against running isolated processes within a single operating system - such as by running multiple JVM instances for Java applications.

In general, multi-tenancy can be achieved at different levels, yet lower the level, the lesser amount of resources different paritcipant can share, and therefore, lesser the cost savings. Guo et. al. [9] categorize multi-tenancy as single instance vs. multiple instance, where the former serves many users using the same instance whereas the latter serves users by running multiple instances using one of the aforementioned methods. Our proposed solution falls under multi-user single instance or native multi-tenancy.

III. Multi-tenancy in SOA

To understand the SOA multi-tenancy problem, let us focus on the high level multi-tenancy architecture proposed by Chang et al. [4] where they identified metadata services, data services, process and business services, security services, and presentation components as different aspects of the architecture (High-Level Architecture section in [4]).

SOA applications also include above aspects apart from the presentation layer. We group above services under Execution, Security, and Data. Typically, execution involves developing, deploying and running services, which are often implemented as Web services, and composing those services together to create higher level artifacts like Business Processes, Workflows and Mashups. Executions capture business and process services defined in Chang et al. Those executions may store and use data either from a registry, or a database, and store metadata in a registry. Security services define the ownership and authorization of both data as well as executions in the framework. As pointed out by Chang et al. [4] as well as by other publications, the goal of multi-tenancy is to provide different users of the system (which we shall call tenants) isolation in each of these spaces while maximizing resource sharing. However, as is often the case, resource sharing and isolation are a tradeoff. Furthermore, Chang et al. [4] has proposed three properties for multi-tenancy in addition to isolation: scalable, multi-tenant-efficient & configurable. Here, multi tenant-efficient means that same instance hosts multiple tenants, a requirement for maximizing resource sharing.

In a multi-tenant framework there is always a risk that the tenant isolation is compromised due to a malfunctioning component or an inadvertent programming error. Furthermore, since the framework is often designed to be extensible, the risks are higher making design and maintenance harder. Therefore, providing an isolated space per tenant/users in the SOA framework is a challenging problem.

An organization that registered as a tenant should be able to manage/administer its own users, data and services. But they should be restricted from administration functionality of the overall application such as shutdown and viewing system logs. Only the administrators of the infrastructure (for whom we use the term super-admins) will be able to access these functions in order to help maintain & operate the running SOA framework. The application core should be aware of handling this two level of access control and component authors should carefully define which functions are accessible to tenants. In other words, the application seen by each tenant should be configurable without compromising isolation.

Finally, since multi-tenancy accumulates requests for many tenants and processes them through a single server, the load received by a multi-tenant server is high. Therefore, elastic scalability, or the ability to bring up additional resources as load increases and the corresponding ability to bring them down, is of paramount importance.

IV. Proposed architecture

This section describes the proposed architecture to implement multi-tenancy for an SOA platform. Since the solution deals with security and implementation related complications, we describe it in terms of a concrete SOA platform. The proposed solution is implemented on top of the WSO2 Carbon platform [10], a componetized open source middleware framework for building scalable, high performance servers. However, most concepts we use are general, and we believe a similar implementation can be done on top of other middleware frameworks as well.

This section first introduces the WSO2 Carbon platform and its architecture. Then we discuss different aspects of the system architecture as it pertains to multi-tenancy for SOA: service deployment, message dispatching, security, service execution, and finally data access.

A. WSO2 Carbon Platform

WSO2 Carbon is a Java based componentized platform for building servers which have the following characteristics:

- they consist of components which fit into the server runtime framework
- they depend on the services provided by the framework for system services
- they offer a Web services API for management
- they offer components which fit into the Carbon administration console to interact with the services and manage themselves

The platform consists of three components: the WSO2 Carbon core, the management console user interface framework, and all other components that execute within the framework.

Core: The WSO2 Carbon core is the main engine of WSO2 Carbon and manages the loading, linking, execution and management of WSO2 Carbon Components. The component model used by WSO2 Carbon is OSGi [11], which is the widely adopted standard for modular components in the Java platform. OSGi "bundles" have a well-defined way to define their published interfaces, indicate external dependencies and manage versions. Because of the dynamic nature of OSGi, all bundles can be loaded/unloaded at runtime and a server can even run multiple versions of the same component at once.

The WSO2 Carbon core offers a set of services for components executing in it. These are:

- authentication and authorization management for administrative services of the components
- authentication and authorization for services offered by the component, including full user management and role based authorization
- shared repository to store state on a per-server basis
- shared repository to store cluster-wide configuration state
- shared registry to store system governance information and to look up data and metadata for runtime governance
- services to share runtime data across nodes in a cluster
- user interface framework for writing UI components (Google Gadgets or regular DHTML applications) to give a human interface to the administrative services of components

Certain other lower level services, such as logging and thread pool management, are inherited from the underlying Java platform.

Components: WSO2 Carbon components are OSGi bundles which offer some functionality into the overall system (and of course may depend on other components in the system). Furthermore, Components can be grouped into the concept of "features", which are logical pieces of functionality that stand on their own. Thus, a WSO2 Carbon based server is basically a collection of features that have been pre-configured to start at the same time. Features can be brought into the server and removed from the server dynamically, using an OSGi provisioning system called P2. This enables a server to customize its personality dynamically to exactly match the environment needed to run a specific function, such as a business process or a Web service.

Management Console: The WSO2 Carbon management console is a componentized Web application built using the WSO2 Carbon core and uses OSGi to manage its components.

The key feature of the WSO2 Carbon management console is the console user interface framework, which provides a template UI for all WSO2 Carbon server management consoles. The template defines how a specific UI component for some feature may integrate with the overall UI - such as where menus fit in and what screen real estate the various pieces go into.

When a WSO2 Carbon management console is pointed to a running WSO2 Carbon server, it detects the features installed in the server and can dynamically reconfigure itself by loading and running only the corresponding UI features in the management console. Thus, the same Web application can be used to manage any WSO2 Carbon server.

Products: A product based on the WSO2 Carbon platform is implemented by composing different features as needed to achieve the desired functionality of the product. In addition, one could start with a bare WSO2 Carbon runtime and keep installing components to form a custom runtime as well. This also enables one to provision a WSO2 Carbon server for an application in a manner that it contains only exactly the functionality needed for that application.

Thus WSO2 Carbon is a composable server architecture which offers an extremely powerful framework for building middleware servers.

B. Service Deployment

Currently, all service hosting components of WSO2 Carbon (including JAX-WS [12] and Apache Axis2 services [13] and BPEL processes [14]), have Apache Axis2 as its underlying execution engine.

The architecture of Apache Axis2 (Axis2) was inherently designed to support multi-tenant deployments. Overall, Axis2 is a totally stateless system with all configuration state stored in exactly one configuration tree called AxisConfiguration and all runtime state stored in another single tree called ConfigurationContext (parts of which are of course only active during the lifetime of one message). All configuration elements such as services & modules are created within the scope of an AxisConfiguration. Hence, the most natural way to think of an Axis2 multi-tenant deployment is to have multiple AxisConfiguration instances; one per tenant. Each tenant's services, modules & other Axis2 deployment artifacts will be created within its own AxisConfiguration.

Axis2 loads execution and configuration artifacts from a repository. In order to make that multi-tenant, we give each tenant a separate repository in the WSO2 Registry (see later for discussion on multi-tenant data management).

In addition to per-tenant configurations, there will also be a master AxisConfiguration, which will be responsible for dispatching requests to tenant AxisConfigurations and provides the management infrastructure for managing tenants. This master (or super tenant or tenant0) AxisConfiguration is created at system boot-up, and belongs to the server. Since tenants cannot 'see' outside their AxisConfigurations, they will not be able to manipulate code hosted by other tenants or change any configurations in the master.

C. *Message Dispatching*

When messages arrive into a multi-tenant WSO2 Carbon server, they always first arrive at the handlers and dispatchers defined by the master configuration. The master configuration will include handlers that implement any cross-tenant policies and SLA (service level agreement) management. For example, if necessary, a priority based dispatcher can be deployed at this stage to offer differentiated qualities of service for different tenants.

When a client sends a message addressed to a particular tenant's service, that request must indicate the tenant in some manner. The default approach in WSO2 Carbon is to add the tenant name to the URL as follows:

`http://example.com/t/tenant-name/services/service-name`

The second approach is to configure a CNAME record in DNS (Domain Name System) as an alias for this information.

Once the tenant is identified, WSO2 Carbon will dispatch that message to the relevant tenant's AxisConfiguration object. Once in the tenant-specific AxisConfiguration, all tenant-specific handlers and modules will be executed, including tenant-specific security and other policy handlers.

Figure 1 illustrates this approach.

D. *Security*

A key challenge in enabling multi-tenancy is providing mechanisms for tenants to have fine grained security control over its services. The requirements include having the ability to have a tenant-specific user base, supporting tenant-specific security policies for services and differentiated QoS based on users and roles.

User Management: User manager represents all permissions using the tuple (role, resource, action) and uses permissions to compute the authority of a user to perform an action on a particular resource. The resource can be an actual resource

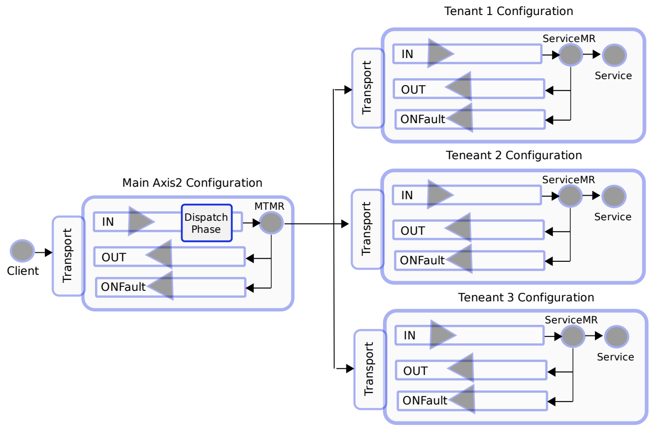


Figure 1: Message Dispatch

in the system registry or a permission. In order to unify the behavior, permissions are also mapped to a registry resource. Each component defines a permission set that restricts users from accessing their features. The user manager is thus able to calculate the authority for features with the same logic that it computes authority for registry resources.

To apply this model in a multi-tenant environment, WSO2 Carbon stores users and the resources (along with the permissions) separately per tenant, similar to the way multi-tenant resources are stored in the registry (see below). Therefore, the WSO2 Carbon platform allows a tenant to have its own set of users. This can be supported using the built-in user store, which has been tenant-enabled using the approach discussed in the data management section, or by the tenant administrator configuring the tenant's user base to their own user store (e.g., LDAP server).

Authentication and Authorization: When messages arrive at the tenant-specific AxisConfiguration, the security handlers deployed there will enforce the desired security policy by invoking the user manager of that tenant to provide user verification and authorization.

The tenant specific authorization manager will calculate the authority by applying the above function to users and resources of the tenants. The tenant adminstrator can customize permission settings from the management console provided with the WSO2 Carbon framework.

Super-tenant Authorization: The user manager not only provides authorization management within a tenant, but it also authorizes the actions that should only be accessible to super adminstrators. This is achieved by defining a set of permissions associated with these specific actions and making them available only in the super tenant space.

E. *Service Execution*

Per-tenant service execution within a single JVM is a challenging task as WSO2 Carbon has to guarantee that malicious code from one tenant cannot, under any condition, access data from another tenant. This extends not only to services but to any other artifact that may be deployed as part of WSO2 Carbon application (e.g., modules and ESB mediators). Total isolation of the deployments of these different clients is a primary & mandatory requirement.

On the management side, each tenant is provided with their own customized management infrastructure & management console. The tenants will not be able to see any services or code deployed by other tenants. Multiple tenants would be able to have services, modules & mediators having the same name, and there is no requirement for these names to be unique throughout the system, unlike in a typical single process user deployment. For example, two tenants, foo.com & bar.com will be able to host two services with the name MyService. These services will have unique URLs such as: `http://example.com/t/foo.com/services/MyService and` `http://example.com/t/bar.com/services/MyService`

Isolation is achieved via several levels of protection. At the lowest level, OSGi uses a multi-level class loader structure. Given the model that WSO2 Carbon uses to load services and other artifacts, code running in one service will have no way to access any code from another tenant.

At the next level, it is necessary to protect OSGi services within the JVM. The OSGi runtime has a service directory which can be accessed by arbitrary Java code to discover other services and interact with them. In the current system, any service which is not deployed as a native OSGi bundle is prevented from accessing any OSGi services by the Java Security Manager. When multi-tenanted WSO2 Carbon supports deploying OSGi bundles, then OSGi security will be engaged to manage authorization for OSGi services within the JVM.

For non-OSGi bundles, standard Java security is used to ensure isolation: A custom Java Security Manager is used to give a sandboxed execution environment, very similarly to what Java applets provide. This permits controlled access to certain sections of the registry, to a specific location on disk, to specific network services and so on.

With this combination of approaches WSO2 Carbon ensures that any Java code loaded and run on a per-tenant basis has no opportunity to misbehave and access data or code from another tenant.

Tenants logic can be executed not only as hosted services, but also in different layers inside the carbon platform. One example is to allow tenants to handle tenant specific data by writing custom handlers inside the registry layer(see below). It also uses the security considerations mentioned above to make sure tenants always run inside their sandbox.

F. Data Access

Data isolation across tenants is an important aspect of making an application tenant aware. The WSO2 Carbon framework provides a registry/repository that abstracts out the relational database layer while supporting a simpler resource oriented API to store and retrieve persistent data. The registry implements tenant-wise data separation logic inside itself to provide a multi-tenant aware registry interface to higher layers.

For example, the registry API is designed to access resources using a path string as a key. A users of the registry API gets the 'registry' object from the session and uses `registry.put(path, resource)` to store a resource and `resource = registry.get(path)` to retrieve a resource.

This simple registry API has been made tenant-aware to give each tenant a (virtual) registry of its own. The tenant information is retrieved from the runtime context (where it was placed by the dispatch and security layers) and used to qualify all data accesses from the database.

Virtualization at the registry layer allow services hosted by the tenant to use the tenant specific registry to persist data. The resources persisted in the registry can be accessible via RESTful URIs, in addition to programatically. In the multi-tenant registry, the URI associated with a resource will contain the tenant name to avoid the resource name conflicts among tenants. For an example a resource of tenant-A with the path "/example/path" will have the url https://host/t/tenant-A/registry/resource/example/path. These urls are protected with HTTP authentication to ensure that only users from that particular tenant have the access to the resources.

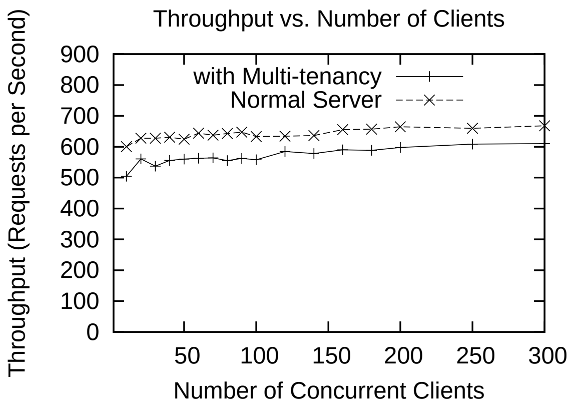
As described in the related work section, there are 3 methods to develop a multi-tenant registry on top of a relational database. They are using a separate database, separate tables, and shared tables. From these methods the first one provides most isolation of data among tenants whereas the last one provides the best utilization of resources. While the first approach it is expected to have higher availability, it comes at the cost of more use of resources, especially for locking transactions. However, since most of the database engines today support row level locking (for example, the MySQL InnoDB engine that we used in our deployment), the third approach can also be expected to have the same level of availability. WSO2 Carbon uses that approach for all data stored in relational databases.

V. Performance Evaluation

In order to enforce isolation, multi-tenancy adds additional code into the execution flow, which of course introduces extra overhead. This section reports on some initial experiments conducted to measure the impact of multi-tenancy on overall system performance using our initial proof of concept code.

For the experiment setup, we use a simple Stock Quote service, which returns a randomly generated stock quote. This service is deployed onto multi-tenanted and non-multi-tenanted versions of WSO2 Web Services Application Server, a WSO2 Carbon based service execution server. These servers are then loaded using parallel clients and message latency and throughput were measured.

Figure 2 depicts the results of the experiment. The test was conducted on an Intel(R) Core Duo 2.4 GHz machine with



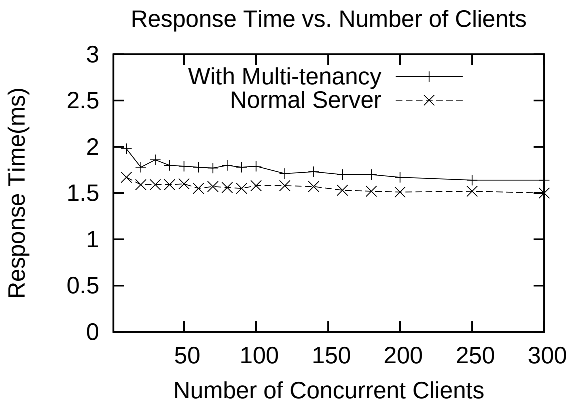


Figure 2: Overhead of Multi-tenancy

8GB memory running Ubuntu Linux. While the test was running, CPU utilization was nearly 90% and the network utilization was around 95%. Furthermore, the machine was able to maintain an almost constant memory heap while increased the load and the concurrency.

As expected, there is an overhead introduced by multi-tenancy. However, the overhead was only around 10-20%, and even with multi-tenancy, the server was able to sustain a respectable throughput of 600 requests per second. Given the improved utilization delivered by multi-tenancy, the initial measurements suggest that the resulting system is effective despite the overhead introduced by multi-tenancy.

VI. Security Review

At the deployment level, we have isolation since each tenant gets its own AxisConfiguration & ConfigurationContext. Further security restrictions are imposed by running tenant code within a JVM sandbox. A Java Security Manager restricts access to the file system & classloaders and restricts operating system interactions such as socket operations, networking operations, systems commands & thread operations. The security manager will be deployed on each server where tenant code is deployed. There will be a small performance overhead introduced by the security manager since some of the operations at the JVM level will always go through the security manager. Furthermore, access to critical data structures and operations are only allowed to internal code, and not to the code deployed by end users. This is enforced through the Java platform mechanisms which enable developers to control access provided to different code segments where the platform identifies the authorized code by verifying signatures of jar archive files that contain them.

In addition to the restrictions the Java security manager enforces, the WSO2 Carbon framework ensures that there are no back doors to access the file system. For an example, in the registry, users can upload resources by providing a URI, but registry will not continue if the provided URI is starting with ”file:” prefix as it refers to a file in the server file system. This prevents users from reading server configuration files and stealing sensitive information such as database credentials.

Services hosted by tenants will not be able to access the database and execute SQL queries directly. This is prevented by making all the core data access packages private in the registry core and user management OSGi bundles. Therefore, tenants can only access the data layer through the registry interface which enforeces tenant isolation. SQL queries in the registry and the user manager are always used with SQL parameters to pass user data to ensure protection from SQL injection attacks.

VII. Discussion

Advent of cloud computing has provided tangible prospects of cutting down the cost of IT infrastructure. However, abstractions they provide are far from the expectations of users. For example, most cloud offerings provide either bare OS or programming language APIs, whereas users would ideally want to host their applications without worrying about intermediate layers. Furthermore, to achieve the economies of scale that SaaS and PaaS proponents have prophesied, different users would have to share much more than the same hardware. Therefore, there is need to fill the gap between what cloud offerings have provided and expectations of IT managers, and multi-tenancy has been proposed to solve this problem.

SOA based applications have a heavy footprint in most IT infrastructures where they are either exposed directly to users or are back-ends for internal applications. In such environments, supporting multi-tenancy at the SOA framework level would enable users to move their applications to a multi-tenant environment with almost changes to those applications. After motivating multi-tenancy for SOA in Section III, this paper presents a design and an architecture of such a multi-tenancy SOA platform.

The proposed architecture is built on top of the WSO2 Carbon platform, which is a componetized middleware framework that enables users to share common aspects like security and data implementation across all the products in the SOA framework. As discussed in the earlier sections, we divide the multi-tenant architecture to three parts: execution, data storage, and security.

In Section IV we discussed in detail how each of these aspects was realized. An initial set of performance measures are presented in Section V to validate the proposed architecture. Then Section VI reviews security implications of this platform.

A key promise of multi-tenancy is improved utilization. In order to achieve optimal utilization it is necessary for the system to scale elastically. In the WSO2 Carbon platform, this is achieved by placing the WSO2 ESB (Enterprise Service Bus) as a dynamic load balancer in front of an elastic cluster of WSO2 Carbon servers. When the load balancer detects that available servers are not keeping up with load, it will automatically initiate another server and have it join the same cluster. Since all nodes in the cluster support all tenants, that immediately increases resource availability for all tenants. Unlike with simple OS virtualization, resources are now completely shared across all tenants and hence optimally utilized.

Yet there are some difficult challenges to be resolved. For example, elastic adjustment in the presence of session-oriented interactions is difficult. In some cases sessions may be replicated across nodes in a cluster but typically that is done for a few nodes only (to prevent session state clustering from using up too much CPU time and network bandwidth). Then when a server needs to be released because load has come down, any sessions in that server must be migrated to another server and the session-aware load balancer suitably notified.

While the multi-tenant data solution WSO2 Carbon provides is sufficient for many scenarios, it is still necessary to provide direct database access in a tenant-safe manner. An approach would be to implement a tenant-aware JDBC driver with choices on how to realize tenant isolation. Direct database access at the JDBC layer would be extremely helpful towards supporting safe execution of arbitrary code in a multi-tenant environment.

Another long term challange in multi-tenancy is how to scale up to global scales. Larger servers and local clusters gets one so far, but one needs massively distributed architectures to support global scale multi-tenancy. Just like many power stations are combined to a power grid and then distibuted to many users, massively distributed architecture could put the power of many servers together and serve many clients. The so-called Hybrid Cloud concept is heading in this direction.

We believe our proposed architecture would be useful for two types of audiences. First, PaaS providers who want to build multi-tenant environments for scalably hosting SOA artifacts. Second, even though we believe PaaS/SaaS providers will be successful, it will be a long time before all local IT management operations are shifted to PaaS/SaaS solutions. Thus, we also target our solutions to organizations who want to consolidate their IT foot print by putting many instances of their SOA artifacts in the same SOA container and optimally operate their data center. Consequently, we believe proposed multi-tenancy platform will be a useful addition to architectures of both PaaS/SaaS providers as well as normal organizations.

References

[1] M. Armbrust, A. Fox, Griffith *et al.*, "Above the clouds: A berkeley view of cloud computing," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28*, 2009.

[2] M. Turner, D. Budgen, and P. Brereton, "Turning software into a service," *Computer*, pp. 38–44, 2003.

[3] M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: State of the art and research challenges," *Computer*, pp. 38–45, 2007.

[4] F. Chong and G. Carraro, "Architecture strategies for catching the long tail," *MSDN Library, Microsoft Corporation*, 2006.

[5] L. Tao, "Shifting paradigms with the application service provider model," *Computer*, pp. 32–39, 2001.

[6] D. Jacobs and S. Aulbach, "Ruminations on multi-tenant databases," *BTW Proceedings*, 2007.

[7] F. Chong, G. Carraro, and R. Wolter, "Multi-Tenant Data Architecture," *MSDN Library, Microsoft Corporation*, 2006.

[8] S. Aulbach, T. Grust, D. Jacobs, A. Kemper, and J. Rittinger, "Multi-tenant databases for software as a service: schema-mapping techniques," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, 2008, pp. 1195–1206.

[9] C. Guo, W. Sun, Y. Huang, Z. Wang, B. Gao, and B. IBM, "A framework for native multi-tenancy application development and management," in *International Conference on Enterprise Computing, E-Commerce, and E-Services*, 2007, pp. 551–558.

[10] "wso2.org/projects/carbon," [Online; accessed 20-March-2010].

[11] O. Alliance, "OSGi service platform, Core Specification release 4.1," *Draft, May*, 2007.

[12] J. Kotamraju, "Java API for XML-Based Web Services (JAX-WS 2.0)," 2006.

[13] S. Perera, C. Herath *et al.*, "Axis2, middleware for next generation web services," *Proc. International Conference on Web Services, 2006*, pp. 833–840, 2006.

[14] T. Andrews, Curbera *et al.*, "Business process execution language for web services, version 1.1," *Standards proposal by BEA Systems, International Business Machines Corporation, and Microsoft Corporation*, 2003.