Cloud Services Gateway: A tool for exposing Private Services to the Public Cloud with fine-grained Control

Srinath Perera, Rajika Kumarasiri, Supun Kamburugamuva, Senaka Fernando,
Sanjiva Weerawarana and Paul Fremantle
WSO2 Inc.
No 59, Flower Road
Colombo, Sri Lanka
{srinath, rajika, supun, senaka, sanjiva, paul}@wso2.com

***Abstract*—By enabling users to allocate computing resources on demand, cheaply, and in an elastic manner, Cloud Computing has made large computation resources available to small and medium size organizations. However, using the Cloud requires users to place their computations, data, or both in a shared data center own by an outsider. This sharing has raised many security concerns. Such concerns are much apparent with use cases like health informatics, where the security of the information is critical and imposed by government regulations. We propose a hybrid approach to solve this problem, where only computations are moved to the public domains while keeping the data within the private network, and computations may access data through a set of services that expose data following the Principle of Least Privilege. Such architectures will, however, require computations in the cloud to connect to the local network that holds the data, and the obvious solution: that is opening up ports in the organizational firewall could potentially create security loopholes. As an alternative, we propose Cloud Services Gateway (CSG), which enable users to selectively expose their private services that reside inside a firewall to outside clients while maintaining fine grained control. This paper motivates hybrid Cloud architectures and presents the architecture and design decisions of Cloud Services Gateway.**

***Index Terms*—cloud computing, security**

I. Introduction

A medical researcher at a teaching hospital, Richard, has detected a correlation between cancer and patient eating habits while processing last six months of data with his personal computer. Richard has access to similar data for last 10 years and like to extend his analysis to that data as well. Talking to his students, Richard found out that processing with a single computer would take 2 years. Talking to his colleague, who is a computer scientist, Richard found out that parallel processing with Map Reduce [10] could finish the processing in days by running the analysis with 1000 nodes. Unfortunately, it is very hard to justify buying thousand computers and maintaining it by a hospital, and few years ago, Richard would have no hope.

Cloud computing would let Richard to finish the analysis in days for few thousand U.S. Dollars. However, data Richard is dealing with are health records, which are sensitive and protected under government regulations, and releasing this data to a public Cloud and processing it there would raise many eyebrows.

Above scenario is a typical example where Cloud Computing has enabled use cases that were once impossible. At the same time, the use case brings into focus Cloud Security, a major concern associated with Cloud Computing. For example, according to Leavitt [14], in a survey of Chief Information Officers and IT executives, 75% of participants has said that they were worried about Cloud Security. Kaufman et al. [13] has provided a detailed discussion of some of security concerns.

Cloud Computing forces users to place their systems, fully or partially, within an administrative domain owned by a third party Cloud provider, which poses several technical, trust, and legal challenges.

Firstly, among technical challenges are reliability and availability, ensuring isolation among multiple users at both execution and communication, and battling privacy issues like avoiding a forth party from masquerading as the Cloud provider. Although, the Cloud has borrowed solutions from security and privacy research, their soundness in Cloud settings are yet to be demonstrated, and more often than not, users have to rely on the word of the Cloud provider and apparent safety of the cloud. On the other hand, even if those measures are indeed effective, still there are many avenues where security can be improved. For an instance, Amazon EC2 [2] does not provide a way to delegate user credentials, and therefore, if a user wants to access the Amazon S3 storage [3] from his EC2 instance, he has to copy his Amazon credentials to the EC2 instance. Consequently, a compromise of an instance or a user mistake (e.g. saving amazon image that has credentials) can potentially compromise the whole system. This specific case can be addressed by using one of the well-known delegation mechanisms.

Secondly, due to the lack of technical means to prevent Cloud providers from accessing systems hosted within their data centers, Cloud solutions forces us to trust the Cloud

Provider. Among criteria of trust would be the credibility of Cloud providers, in terms of their size, process, and traceability of their operations etc. However, given a credible provider, the required amount of trust is decided by the risk associated with the use case. As shown by Figure 1, there is a spectrum of use cases where one end consists of use cases that manipulates public data, hence have very low risks associated with them, while the other end consists of use cases that manipulates credit cards, Social Security Numbers, or ultra sensitive data like nuclear tests. For the low end of the spectrum, Cloud Computing is an obvious choice, while for the high end, Cloud Computing might never be a choice.

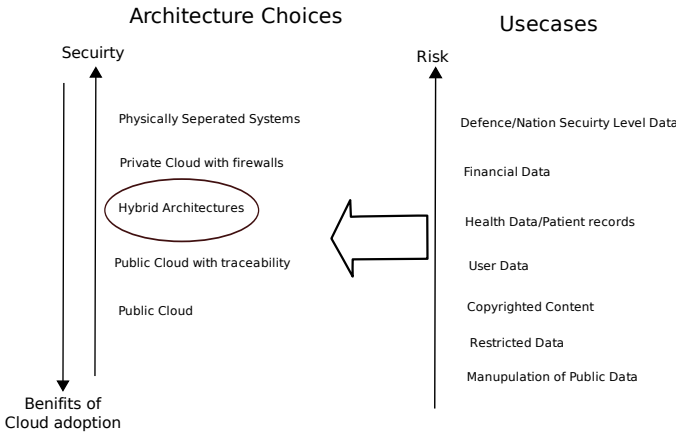

Fig. 1. Risk Spectrum and Architecture Choices

Instead of public and private cloud, Hybrid architectures provide another choice, a middle ground. For example, hybrid architecture could move computations to the cloud while keeping the sensitive data in a secure database that resides in the private network. A set of secured services could expose that data to computations while following the principal of least privilege, and at the public cloud, the computations should only keep that data in-memory and should never write them to a persistent storage. Furthermore, it worth noting that there exists tools that support the development of such services. For an instance, Data services [19] have enabled implementation of such services without writing any code at all, purely through configurations.

However, it is worth noting that even hybrid systems would not solve cloud security concerns completely. For example, administrators in the services hosting company can still use tools like Xenaccess [9] and get to the data held within the memory. However, this model reduces the chances of exposing data by a mistake, and makes the attackers work much harder than in the case of data placed in the cloud. Furthermore, since the data still resides in the private network, it addresses some of the legal concerns as well.

Consequently, although hybrid architectures are not a silver bullet for all use cases, we believe they provide a very important design choice in the risk spectrum, and worth considerations with many Cloud use cases. Figure 1 also shows some of the architectural choices and the place hybrid architectures hold among them. It is important to note that although the figure shows risks and architectures side by side, a direct mapping from security risks to architectures is not known, and the research community is only working towards figuring out those mappings.

Such hybrid architectures typically export selected data products as data services and compose those services to implement the overall computations. For computations to access the data that resides in the private network through services, at least few ports in the organizational firewall need to be opened. However, opening ports might open up security loopholes that could be exploited by attackers leaving the private network vulnerable.

As a solution to this problem, we propose “Cloud Services Gateway”, which can open up private services that reside inside a firewall to outside clients in a controlled manner without opening any ports in the private network. As we shall discuss in Section VII, there are several alternative approaches that can be used to solve the same problem. However, as Section VII illustrates, CSG provides a transparent—meaning that users do not need to change existing applications– and fine-grained control in comparison to other solutions. Our contributions of this paper are two fold.

1) We motivate hybrid systems composed of multiple security domains as a potential solution to Cloud security concerns.
2) We proposes CSG as an enabler of hybrid systems, which enables users to expose their services to the public cloud without any changes and in a push of a button, Furthermore, CSG enables them to maintain fine grained security control over exposed services.

In the following discussion, we shall use the term “Cloud Service” to refer to a private service which is exposed to the public Cloud using the gateway. Furthermore, we shall use the term “Web Service Container” to refer to a middleware framework that facilitates users to develop, run, and manage Web Services implementations. Apache Axis2 [4] and WSO2 Web Services Application Service(WSAS) [7] are examples of such Web Service Containers.

The next two sections will discuss the problem and the proposed solution in detail, which is followed by a description of its implementation. Section VI presents a security discussion of the proposed solution and an empirical analysis of the overhead induced by the solution. The following section discusses related works, and finally, Section VIII concludes the paper.

II. The Problem

As discussed in the introduction, hybrid architectures provide a useful option for mitigating Cloud Security concerns. Within Hybrid architectures, some parts are located in a public Cloud while the other parts, like data, are located in a private network. In a typical organization, the organizational firewall secures the private network and stops any incoming networks, and therefore, computations in the public cloud cannot access services that expose data stored in the private network. This

paper proposes a middleware framework that uses pooling to address this concern.

However, as we shall discuss shortly, it is relatively easy to work around the firewall by writing custom code, but that would call for changes to existing applications. For example, many systems work around the firewall by placing a server outside the firewall to receive messages and asking entities inside the firewall to use polling to receive requests from the server. However, blind application of this solution will require changes to each application, and therefore, we believe enabling Cloud support without affecting existing services is a key requirement.

To recap the requirements, we argued that by supporting hybrid systems that span multiple security domains, we could address some of the Cloud security concerns. To that end, we need a virtual connection from outside of the firewall to services inside the firewall, which selectively (with fine-grained control) and in a secure manner, exposes existing services to clients inside other security domains while requiring minimal changes to the existing services.

III. Proposed Solution

One could send a message from the outside of the firewall into a system that runs inside the firewall or in other words, to work around the firewall, by using a reverse transport. A reverse transport includes a server outside the firewall that receives request on behalf of the real receiver inside the firewall, and a cooperating client at the receiver. This client either periodically polls the messages from the server (e.g. Caromel et al. [18] and some JMS implementations) or the client connects to the server and keeps a connection open, which is used by the server to push new messages message back to the client. For example, XMPP chat clients [16] that reside in a firewall create a connection to the XMPP server and keep the connection alive so that the server can push messages into the client by writing data to the connection.

However, users of cloud services should need to know about the reverse transport or aforementioned mechanics to push messages into the firewall. Hence, the external server that receives messages on behalf of the cloud service should expose the same service interfaces to the end user. To do this, we use Service Proxy pattern supported by Enterprise Service Bus [17]. Furthermore, this approach only let service calls targeted to the exposed service with proper Service formats (e.g. SOAP or REST) through, and this provides added security in comparison to opening up ports or creating a SSL tunnel from a public Cloud to a private network.

Figure 2 depicts how CSG puts together above two aspects of the solution together. An Enterprise Service Bus (ESB) that resides in the public Cloud builds a connection between private and public Cloud using a reverse transport. We shall call the Enterprise Service Bus as "Gateway" henceforth. Gateway exposes a Proxy service for each published cloud service, and when a proxy service receives a message, the proxy service routes the message to the actual service implementation that resides in the private network through the reverse transport.

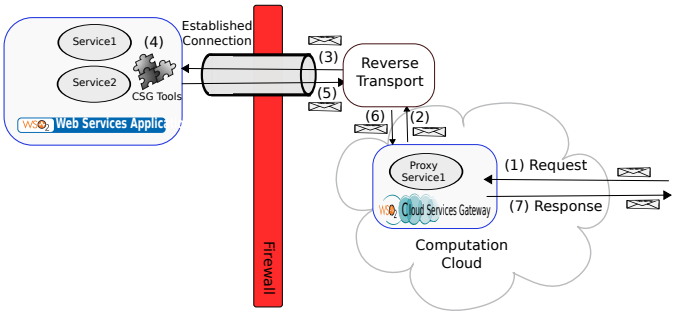

Fig. 2. High Level Architecture

A user can implement this scenario using a ESB by changing the service implementation to poll the reverse transport. However, that would force him to do that work for each service he want to publish. We have developed a code (agent) to simplify the user experience further, where the agent can integrate with existing Web Service Containers, and on request of the user to automate the creation of Proxy services. The agent will inspect local services, extract required settings, create a proxy service, and configure the proxy service to talk to the backed service implementation through a reverse transport. Consequently, a user can expose his pre-developed service to the public cloud, just by pointing his Web Service Container to the gateway, selecting the service, and clicking a button. Agent handles all the details, and in the next section, we shall dive into the implementation in details.

IV. Gateway Implementation

As described in the previous section, the gateway architecture includes an Enterprise Service Bus that acts as a Gateway, a Web Service Container that hosts the actual service implementations, and a reverse transport that is setup between the gateway and the Web Service Container. When a user requests to publish a Service to the public cloud, a CSG clients that runs in the Web Service Container creates a Proxy service within the ESB for each published service, and those proxy services are configured to use the reverse transport to communicate with the real services in the private network. Following five subsections present design decisions and the workings of CSG in detail.

A. Choice of reverse transport

As described earlier, a reverse transport is a polling based delivery method that is capable of sending messages from outside of the firewall to a service running inside the firewall. We can implement such a transport through several methods. In the following discussion, we will call the host where the cloud service implementation resides as the "implementation" host and the host where the gateway resides as the "gateway host".

One option is XMPP, where the service, which acts as the XMPP client, initiates and keeps-open a TCP connection from itself (private network) to the XMPP server (public network),

and the XMPP Server sends messages to the client by writing to that connection, thus pushing the message in to private network.

Another alternative implementation is to use a SSH tunnel to map a port in the gateway host to a port in the implementation host. Since the firewall only allows outgoing traffic, this connection must be initiated from the implementation host. When the gateway receives a message, it will write the message to the local port, and SSH tunnel will transfer messages to the application that resides inside the firewall. However, we have implemented this and found out that the SSH tunnel drops connections time to time, and this typically happens due to network glitches. Moreover, when the connection drops, SSH tunnel does not reconnect automatically and communication between services and gateway stops. Therefore, we have not used the SSH based solution.

Also, Java Messaging Service (JMS) [12] can be used for the similar effect, where a JMS broker runs in the public network and a JMS client running inside a firewall polls the JMS broker and receives messages.

Another method for the implementation is to use a similar technique as the JMS implementation. Apache Thrift framework [5] based server can be used as the broker that runs in the public network and the same framework based client can be used within the firewall to poll the broker.

Finally, it is also possible to implement the reverse transport by setting up a Virtual Private Network (VPN), or in other words, with a VPN, a reverse transport is not necessary. We will discuss pros and cons of this approach in the related works section. As we shall discuss in the security discussion, to ensure security, we need the Web Service Container to authenticate with the messaging server it uses to send the messages (e.g. XMPP Server, JMS broker or the Thrift server).

B. Overall Architecture

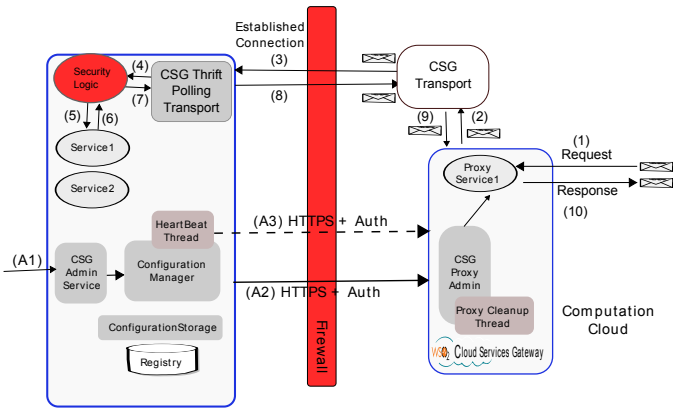

Fig. 3. Gateway Implementation

We have implemented the Gateway using XMPP, JMS, SSH and Apache Thrift framework. However, for the final implementation, we used the Apache Thrift based reverse transport. This implementation was selected after a number of iteration over the JMS based implementation mainly due the performance gain. The JMS implementation also tested with number of JMS brokers such as HornetQ [6] Apache Qpid and Apache ActiveMQ but the Apache Thrift based reverse transport displayed the best performance. WSO2 ESB [8] was used as the Enterprise Service Bus, and WSO2 WSAS [7] used as the Web Service Container.

A user may send SOAP messages to CSG server through protocols like HTTP, SMTP, JMS, and XMPP, and in Apache Axis2/WSO2 WSAS terminology, we call them as transports. For example, the JMS transport enables users to send and receive SOAP messages to or from other SOAP containers using a JMS server to communicate SOAP messages. Then CSG sends those messages to the actual cloud service implementation using a reverse transport.

C. Apache Thrift based reverse transport architecture

The Thrift based reserve transport consists of a Thrift server running within the CSG, which is outside of the firewall, and a polling Thrift client running in the WSAS, which is inside a firewall. The Thrift server runs in the same JVM as the CSG server, and it supports two Thrift operations: login() and exchange(). The exchange operation let clients to retrieve any pending messages intended for the application server and delivers any processed responses to the Thrift server outgoing queues. The login operation enables users to initiate the connection and receive a token, which they should use with exchange() invocations.

When the CSG server receives a request, it sends the message to reverse transport as we discussed before. As shown by Figure 4, the reverse transport simply copies the message into a queue insides Thrift server. Each application server that has published a service in to the cloud using CSG has a client that periodically invokes the exchange() method as a Thrift call, retrieves any pending messages intended for the application server, and delivers any processed responses to the Thrift server outgoing queues. Hence the intended client picks up the message from the Thrift server, processes the message by invoking the real service implementation, and sends the response back to the Thrift server using the next exchange() operation invocation.

When CSG receives a request, the original processing thread blocks while the message placed on the queue of the Thrift server is delivered through the reverse transport, processed, and response is being sent back. When invoked, the exchange() operation signals the threads associated with the messages for which the response has been received. Also the thread blocking has a timeout to avoid threads from blocking forever if some failure prevents the response from coming back.

We have implemented this from the scratch to achieve the maximum performance. Compared to JMS, the Thrift implementation allows us to batch many requests in to one message exchange and also allows us to do retrieving requests as well as delivering the responses also within a one invocation. As we shall see from the results, the Thrift based reverse transport was able to increase the performance from 50% (with JMS) to 94%.

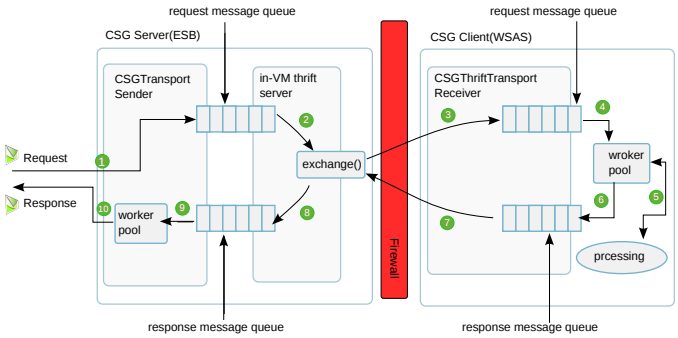

Fig. 4. Reverse transport architecture

D. Publishing as Proxy Services

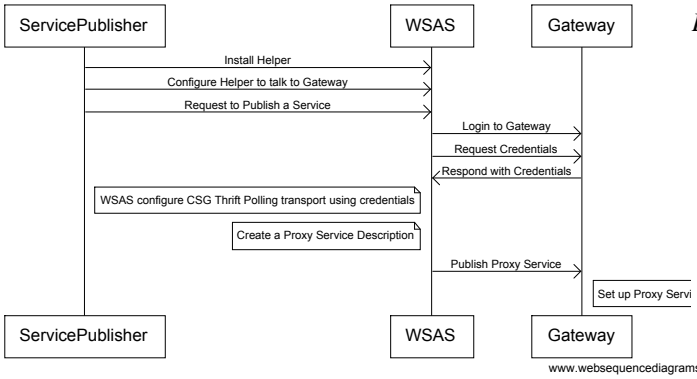

Fig. 5. Steps in Publishing a Proxy Service

Users use CSG to publish their services running within their Web Service Container (e.g. WSO2 WSAS) and within the firewall securely to the cloud. When published, end users may send messages to the gateway, and the gateway will route the messages to the real service and send results back to the end user. To understand CSG, let us look at the working of publish and message routing.

Figure 5 depicts the publish process. Before publishing, user should add CSG client to his Web Service container (WSAS) and then configure the client pointing it to a CSG gateway instance. When this done, user may publish a service by selecting service from a list and pushing a button.

When user publishes a service, CSG client prompts the user for credentials of the CSG, and using those credentials, the client creates a proxy service for the service to be published and configure a reverse transport through Apache Thrift. For this process, WSAS(CSG client) has to authenticate itself with the ESB server(CSG server). When user asks to publish the service with the given credentials CSG server first check the permission for the credentials and if required permissions are given will respond back to the CSG client with a secure random Integer. This Integer acts as the token to access the servers request messages. Then in the next steps CSG client configures the CSGTransport(at the CSG server side) and the CSGThrift polling transport(at the WSAS side) thus setting up a reverse transport. Finally, the CSG client inspects the service to be published, creates a description of a proxy service that would reroute requests targeted to itself to the actual service implementations that resides in the firewall, and deploys the proxy service to the gateway. The same token is pass to the proxy service so the proxy service also be able to put messages into the Thrift server.

Apache Thrift based reverse transport uses the above token based method for authorization. However, the same was not true with XMPP, and we supported XMPP scenario in the following manner. When a service is published with CSG through XMPP, the CSG client talks to the CSG, which creates new XMPP accounts and sends them back to the client, and the client uses those accounts to configure the XMPP transport that talks with the XMPP servers.

E. Invocations

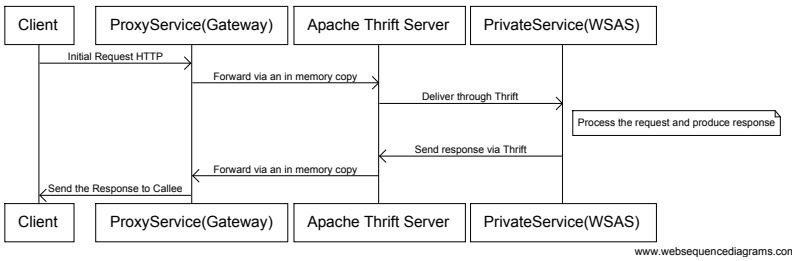

Fig. 6. Steps in invoking a Proxy Service

Figure 6 depicts how CSG works at the runtime, and it shows how the gateway exposes private services to the public cloud by rerouting requests received by proxy services to the service implementation that resides inside the firewall. When a new request arrives at the proxy service through HTTP or HTTPs, the proxy service accepts the message, addresses it to a queue(that reside within the Thrift server) that is mapped to the backend service, and forwards the message to the queue using an in memory copy using the CSGTransportSender. Then the CSGThrift polling transport in the WSAS Web Service Container periodically polls the Thrift server queue via exchange operation and retrieves the message over an SSL/TLS connection, and by doing so CSGThrift polling transport uses polling to work around the firewall. When the CSGThrift polling transport of WSAS receives the message, it invokes the service implementation, which actually processes the message, and responds back using SOAP message addressed to the gateway. CSGThrift polling transport forwards the message to gateway through an SSL/TLS connection, which will forward it to the proxy service. The proxy service sends the response back to the caller who invoked the service at the first place.

JMS transport also works in the same fashion, where the JMS client runs within each service container periodically poll the messages from the JMS Queue, process those messages by

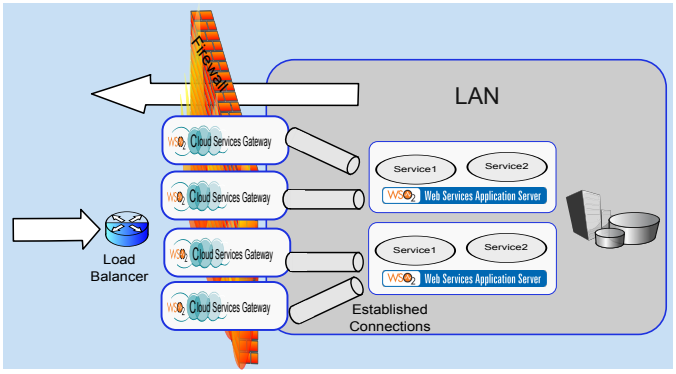

Fig. 7. Large Scale CSG Deployment

invoking the service implementation, and send them back to CSG using JMS.

Alternatively, if XMPP is used, the routing works much similar except for the fact that when WSAS starts, XMPP server creates a connection to the XMPP server from private network and keeps that connection open, and XMPP server uses that connection to push messages to the XMPP transport. In this fashion, XMPP uses this open connection instead of polling.

V. Large Scale Deployments

The connectivity provided by the gateway from the cloud to the private network is stateless, and therefore, it allows multiple CSG instances to run in parallel without sharing any state across the gateway instances. On that setup, users may publish the same service to multiple gateway instances, and each will route messages to the same service implementation. Figure 7 shows a typical large-scale deployment that uses the concept. Since gateway instances do not share any state, this leads to a so called Shared Nothing Architecture [20], which can be scaled up by running enough instances. On such a setup, the system is only limited by the scalability of the services exposed by the CSG, not by the gateway itself.

VI. Evaluation

A. Security Discussion

This section discusses the security model of CSG in details. The Gateway operation consists of two main functions: publishing private services as cloud Services and routing invocations received by published services to and from their actual implementations. Let us consider each.

1) Service Publications: Let us first consider publication of a service to the gateway. As depicted in Figure 5, the client requests his Web Service Container to publish his service as a cloud service. Consequently, the Web Service Container initializes a HTTPS connection to the gateway utilizing the well-known Public Key based Secure Socket Layer (SSL) security [11] and then logs in to the gateway by authenticating himself using a user name and password pair. When requests WSAS to establish the connection, WSAS prompts the users to enter his username a userand passwords for the gateway, and it is assumed that the user has obtained those passwords in an out of bound yet secure means. Moreover, the gateway must have a SSL certificates that are trusted by WSAS, and that protects the users from a third party masquerading as the gateway. Once authenticated, WSAS uses the same HTTPS channel to carry out rest of the interactions described in Figure 5. Since HTTPS provides privacy and avoids man in the middle attacks, those interactions are secure. At the end of the interaction, the local service has been securely published to the cloud through the gateway. The Gateway associates the published cloud service with the publishing user, and before editing or deleting a published service, the gateway verifies that logged in user is the owner of the service. Therefore, only the owner may make changes to the service.

2) Service Invocations: Secondly, let us consider the security while invoking cloud services through the gateway. It is important to note that for cloud services requests, the gateway only ensures a secure connection from CSG to WSAS, which runs inside the private network, and the gateway provides neither authentication nor authorization. WSAS should authenticate and authorize the requests. This is a conscious decision we took while designing the gateway, as providing authentication or authorization support at the gateway would have required WSAS to share user credentials with the gateway. Since the gateway runs in a public network, it can be compromised, and in such a case, sharing such credentials with the gateway will compromise the credentials as well. Hence, the gateway provides only a secure connection between the public cloud WSAS, and WSAS which resides in the private network, provides the authentication and authorization.

Let us explore how this approach would work with SOAP messages secured with WS-Security [15] and messages secured with HTTPS, which are the two main approaches to implement Security in the SOA world.

Supporting end-to-end communication where the message is forwarded by an intermediary is a primary scenarios covered in Web Services. Therefore, if invocations use WS-Security headers, even if the gateway forwarded the message acting as an intermediary, the corresponding WS-Security scenario would work seamlessly. This is possible because with WS-Security, all the information is contained within the SOAP message and hence the fact that message is forwarded by an intermediary (CSG server in this case) does not affect security processing at all.

However, this is not the case with HTTPS requests. When the gateway receives the messages via HTTPs, and the gateway communicates with WSAS through a reverse transport. Since the HTTPS only provides point-to-point connections, the gateway cannot extend the same secure channel to the container. Therefore, there will be a SSL connection from user to the gateway, and there will be additional SSL connections from the gateway to WSAS. For example, with Thrift there will be one SSL connection from Thrift server (that runs within CSG) to WSAS. With XMPP or JMS there will be one SSL connection from CSG to XMPP /JMS server and another

one from XMPP/JMS server to WSAS. When the container (WSAS) has finished processing, the response message travels back through the same SSL connections.

When using XMPP and JMS implementation the gateway accepts response messages only from XMPP or JMS server; hence, there is no chance of someone else to inject a wrong response. However, when the gateway receives a message and sends it to WSAS through XMPP or JMS, before the message returns, someone else (another user who also has an account on the gateway) can send a response message to the gateway's XMPP/JMS address. Although, the session ID in messages will stop false messages from correlating with the original message, as an added precaution, we also verify the sender of response when the response is received at the server. Since the XMPP server mandates that user login through a SSL connection using username and password before sending a message through itself, a user can only send messages under his own name.

When using the Apache Thrift based implementation, each client must produce a secure random Integer when performing the exchange operation. The token is issued at the initial stage publishing the service as a cloud service, and WSAS must remember that token and reproduce it with every exchange call.

Furthermore, due to these two(or three) SSL connections, some information like connecting user's Distinguished Name (DN), credentials send as HTTP headers, Cookies, and any custom HTTP headers will not each the container. To circumvent this problem, we have copied all the HTTP headers and connecting user's Distinguished name to the message send from the gateway to WSAS. Therefore, the container can make authentication and authorization decisions using that information copied from the users HTTPS connection.

The proposed approach is somewhat susceptible to Denial of Service Attacks (DOS). Since the authorization happens in the backend, even bogus requests incur costs on the gateway and backend. However, if you consider a typical message mediation through ESB, which is a well known pattern in SOA, the situation is same. There are few potential solutions to this problem. One option is using standard DOS precautions like throttling and keeping blacklists to drop attack messages as soon as possible at the gateway. Another option is mandating users to have an account in the gateway before sending messages through it (currently, only service publications need a user, not sending messages through the gateway to a service), which enables the gateway to drop the messages earlier.

To summarize, all communications are secured via TLS, which provides privacy and avoids man in the middle attacks. However, since the communication channel is broken to two SSL channels when using XMPP, JMS or Apache Thrift based server, we verify that response is indeed sent by the target receiver to avoid possibility of someone else injecting a bogus response by sending it to XMPP/JMS/Apache Thrift server before the real response. Therefore, that case also provides privacy and avoids man in the middle attacks. Consequently, the gateway provides a secure channel for users to interact with the service implementation that resides in a private network.

Finally, when a service runs in a public network, a compromise by an attacker is always possible. Therefore, let us explore potential hazards imposed by such a compromise. As we discussed, the gateway is only a secure tunnel and backend imposes security, and therefore, even if the gateway is compromised, the intruders will be still blocked by the backend security. Furthermore, since communications happen as Web Services, intruders cannot do anything other than sending a Web Service call to the backend service. However, by compromising the gateway, the intruders can potentially intercept the messages coming to gateway and going out of the gateway. Although, this could be critical if the nature of communication is of sensitive nature, we believe this is much better than with VPN, where a compromised public machine will place the intruders within the private network.

B. Performance Evaluation

Security solutions often have adverse effects on the performance, and therefore, we have performed the following experiment to understand performance implications of the proposed solution. The experiment compares invocations of a Web Services through the CSG against direct invocations of the same Web Service, and it tries to obtain an understanding of the overhead introduced by the gateway. To that end, we invoked a Web Service directly and through the gateway with 1, 5, 10, 15, and 20 parallel clients. To generate the load, we used Java Bench, a Java port of Apache Bench [1]. For each reading, we sent about 5000-10,000 requests to the Web Service and measured the throughput and average Latency.

We have tested the setup using three reverse transports implementations: XMPP, two JMS implementations (Apache Qpid, JBoss HornetQ), and the Apache Thrift based implementation. Tests were run using a 2.4GHz Intel(R) Core(TM)2 Duo machine with 8GB memory running Ubuntu Linux.

XMPP based reverse transport only provided about 20% of the direct invocation performance. Apache Qpid did poorly, only doing about 50 transactions per second (which is about 10% of direct invocations), and taking up to 400ms per request. HornetQ did much better by achieving 50% of the direct calls performance, both in throughput (300 vs 500) and response time (35 vs. 64 ms).

In both cases, using a reverse transport based model based on XMPP or JMS incurs a significant overhead where the cost of XMPP and Qpid were severe while HornetQ did much better. Nevertheless, even with HornetQ, we only achieved about 50% of the direct throughput, which was not sufficient.

Therefore, we have implemented the Thrift based reverse transport, which enable us to optimize by batching messages efficiently into single invocation, and Figure 8 describes the overhead introduce by the Apache Thrift based reverse transport implementation. In Figure 8 Number of Concurrent Clients is the no of concurrent clients that invoke the public service simultaneously and number of Thrift clients represents the configured number of Thrift clients in the Thrift transport at WSAS.

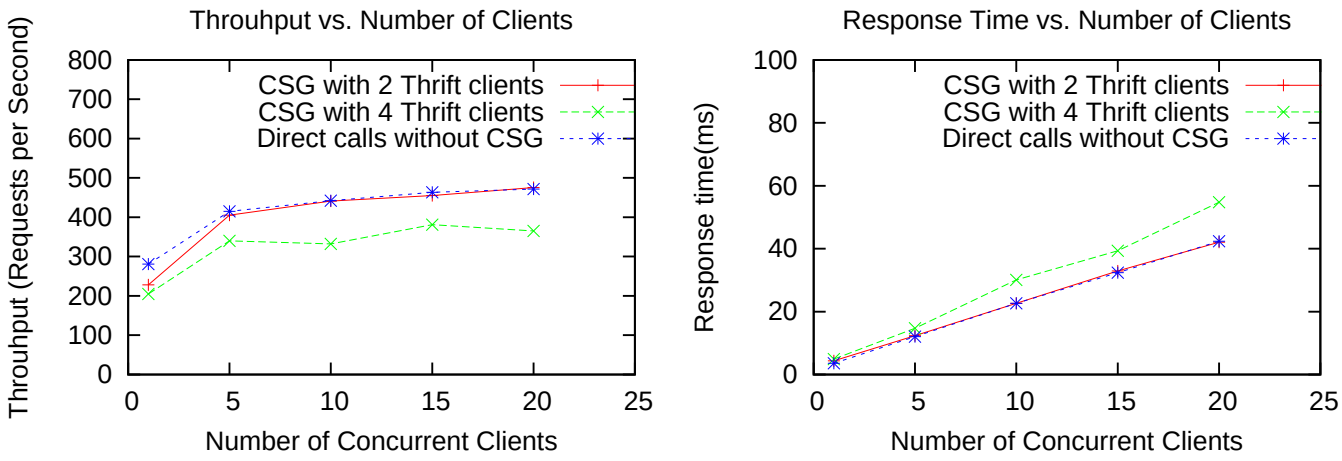

Fig. 8. Overhead Induced by the Gateway

We tested the same scenario with multiple Thrift clients at WSAS(the polling transport at WSAS can be configured with required number of Thrift clients to use) server to poll messages from the Thrift server, and two Thrift clients provided the best results. When using two Thrift clients the hardware was fully saturated under the exposed load and using more than two Thrift clients seems to add more overhead into the system which leads to a degrade in performance compared to using two Thrift clients. When using one Thrift client the hardware was not saturated under the same load. These data showed using two Thrift clients provided the optimal performance under the given load and hardware. As shown by Figure 8, Thrift based reverse transport was able to provide around 94% of the direct call performance. We believe this result is significant, and this will enable the CSG to used in many real world use cases.

We are currently exploring the possibilities of improving the performance. One possibility is to transfer the request messages to the Thrift server at the byte level without parsing message at all. In this scenario the CSGTransport at the CSGServer side just act like a bi-directional pipe where he sends what ever the request/response message to the other side. And also another improvement is to signal the Thrift client when data is available so that Thrift client at WSAS side does not need to periodically polls the server queue for messages. This will decrease the overhead introduce by the polling of server queue in a busy loop.

VII. Related Works

Caromel et al. [18] was one of the first discussions found in literature on the topic of using pull based approaches to circumvent firewalls. In this approach, to communicate with a service inside the firewall, services outside the firewall send messages to a message box, which is also placed outside the firewall. The message box stores those messages and the services inside the firewall periodically polls the message box and process the messages it found in the message box. Although, this approach works, it needs the services living inside the firewall to be aware of the message box and to poll explicitly. In contrast, the gateway works transparent and developers of services do not need to be aware that the services are going to be exposed as a cloud service.

Another alternative is setting-up a virtual private network (VPN) between the public cloud and the private machine, thus placing all parts of the hybrid system in the same virtual network. VPN is an overlay network implemented on top of other networks using cryptographic tunnels for all communication and, therefore, provides a secure network. Although, it is a viable solution, VPN has several shortcomings.

Part of the VPN resides in a public network, and there is always a risk that those public machines will be compromised, which would open up the whole VPN and the local machines to the attackers. Furthermore, since VPN security applies to a connection (e.g. VPN either allow the connection or disallow it, and there is no control over what a user could do with that connection.), it only provides coarse-grained security. In contrast, since CSG uses a Web Services level tunnel, it can enforce much finer-grained control. For instance even if the connection is compromised, the attackers ability is limited to few web services invocations, and hence, attackers cannot use the connection to attack machines inside the private network through that connection.

Furthermore, typically, setting up a VPN and ensuring security is a complex and time consuming process, and if the requirement is urgent and for a short time period (e.g. consider our use case discussed in the introduction, where they need a hybrid system only for few days), setting up a VPN is not feasible and too expensive. In contrast, the proposed gateway only takes few minutes to setup, and it is a very attractive solution for short time use cases.

Finally, an obvious question is “why not open ports?”. As with the case of VPN, when opening ports, it is not easy to limit what can be done through that connection, and hence users will have much coarse-grained control compared

to the fine-grained control provided by the Web Services tunnel. Moreover, often, network administrators will not open ports easily—due to obvious security concerns—and therefore, process of opening up the ports could take days or weeks. Moreover, there are many practical difficulties. For instance, most private networks use private IP translated through NAT, and with that model, opening up ports does not work, and the target machines need a public IP, which may need major changes from network administrator's side.

Therefore, we argue that in comparison to aforementioned solutions, CSG provides a transparent Web services tunnel that provides much finer-grained control over what users are allowed to do using the tunnel.

VIII. Discussion

This paper presents CSG, which can open up private services that reside inside a firewall to outside clients in a controlled manner without opening any ports in the private network. As we discussed in the introduction, the gateway can be used to build hybrid systems that has some of their components in the public cloud and other parts inside the private network that is secured by a firewall. In the previous sections, we have presented the architecture of the gateway and a detailed discussion of its implementation using well-known building blocks from the Service Oriented Architecture (SOA) world.

To work around the firewall, the gateway uses either polling or a pre-established connection initiated from within the firewall. Although, the idea is simple, either would require developers to change their applications accordingly, which is expensive. In contrast, Cloud Service Gateway (CSG) expose existing services to public cloud on a push of a button, without end-user having to understand any of the inner working of the gateway.

Furthermore, we observed that since the gateway extends the security model of the system it is deployed in, the soundness of its security is a key decider of the value of this work. Therefore, in sections VI-A and VII, we have presented a detailed security discussion of the architecture, while comparing and contrasting the proposed architecture with alternative approaches.

As we observed in the discussion, a key advantage of the gateway over other approaches is fine-grained control of what to allow and what not to allow through the gateway. Furthermore, we observed that even if a public part of the gateway is compromised, the potential doomsday scenarios are much manageable compared to a breach of VPN or a system with opened ports.

Moreover, often security solutions have adverse effects on the performance, and we have, therefore, performed a performance evaluation to understand performance implications of the solution. As we observed in section VI-B, adding CSG has a small performance implication. We are exploring causes for this overhead, and if required, its limits can be potentially improved by setting up multiple Gateway instances to route messages from the public cloud into the private network.

Another interesting aspect of this discussion is passing large files as inputs to the cloud services exposed through Cloud Service gateway. Smaller files can be handled by passing them inline in the SOAP message using SOAP with attachments specifications. However, with data intensive use cases–especially in E-Science use cases—using SOAP with attachment is inefficient for transferring large files. Interestingly, if services need to access data in the Internet, the problem does not arise as often firewalls allow outgoing connections. However, if services need to access data placed inside another security domain, the problem needs special handling. For example, upon request for the file, gateway can ask an internal service to transfer file to gateway host (gateway sits outside the firewall and firewall allows outgoing connections) and pass the link to that file client in secure manner. However, that forces gateway to store the file into the file system, which is an added security concern.

Finally, lets us briefly look at potential use cases of the gateway. As we discussed in the introduction, building hybrid systems that host computations in the cloud while accessing the data stored in private networks through services following the "Principle of least privilege" is indeed the motivating use case for the gateway. For instance, that use case can be extended to many use cases both within and outside E-Science world. For example, a large financial analysis can use the gateway to run the computation in the cloud while keeping the data safely stored in the private network. The idea can be extended to many types of data that should be protected from external access.

Moreover, two groups of researchers can collaborate by calling services running in other groups security domain through the gateway. Furthermore, the gateway can be a way to share data between multiple parties in a controlled manner. For example, lets assume that a biochemist wants to share some data sets with his colleagues in a different university. He could create a service-based data Query interface to expose data, and expose the service using the gateway without opening firewalls.

Another interesting related aspect is that, setting up a Could Services Gateway is simple, and for someone who has a Web Service running in his Web Services Container(WSAS), it will only take few minutes. Moreover, we envision public CSG instances that are hosted by organizations, which would allow users to share their local services in a click of a button either free or per-pay basis. On this setting, the gateway would be a great tool for demos and short time interactions. For instance, if two developers are working on a Web Services Interoperability session, they can use the gateway to share their local services with other parties.

Based on the above discussion, we argue that the CSG could be a very useful tool in the architect's toolbox, which will open up a path for new hybrid architectures that span both the public cloud and private intranets. Resulting hybrid architectures provide a powerful tool for handling use cases from the risk spectrum we discussed in the introduction.

Our contributions of this paper are two fold. Firstly, we

motivate hybrid cloud architecture that include of multiple security domains as a choice while addressing cloud security concerns. Secondly, as an enabler of hybrid systems, we proposes CSG, which enables users to expose their services to the public cloud in a push of a button and enable them to maintain fine grained security control over exposed services.

REFERENCES

[1] ab - Apache HTTP server benchmarking tool. *http://httpd.apache.org*. [Online; accessed 20-Sept-2011].

[2] Amazon elastic computation cloud (ec2). *http://aws.amazon.com/ec2/*. [Online; accessed 20-Sept-2011].

[3] Amazon simple storage service(s3). *https://s3.amazonaws.com/*. [Online; accessed 20-Sept-2011].

[4] Apache Axis2 Project. `http://axis.apache.org/`. [Online; accessed 20-Sept-2011].

[5] Apache Thrift. *http://thrift.apache.org*. [Online; accessed 20-Sept-2011].

[6] HornetQ. `http://www.jboss.org/hornetq`. [Online; accessed 20-Sept-2011].

[7] Web Services Application Server (WSAS). `http://wso2.org/projects/wsas/java`. [Online; accessed 20-Sept-2011].

[8] WSO2 Enterprise Service Bus. `http://wso2.org/projects/esb/java`. [Online; accessed 20-Sept-2011].

[9] Xenaccess documentation. *http://doc.xenaccess.org/*. [Online; accessed 20-Sept-2011].

[10] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communication of the ACM*, 51(1):107, 2008.

[11] A. Frier, P. Karlton, and P. Kocher. The SSL 3.0 protocol. *Netscape Communications Corp*, 18, 1996.

[12] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. Java Message Service Specification. *Sun Microsystems, http://java. sun. com/products/jms*, 2000.

[13] Lori M. Kaufman. Data security in the world of cloud computing. *IEEE Security and Privacy*, 7:61–64, 2009.

[14] N. Leavitt. Is Cloud Computing Really Ready for Prime Time? *Growth*, 27:5.

[15] M. Naedele. Standards for XML and Web services security. *Computer*, pages 96–98, 2003.

[16] P. Saint-Andre. Streaming XML with Jabber/XMPP. *IEEE internet computing*, 9(5):82–89, 2005.

[17] M.T. Schmidt, B. Hutchison, P. Lambros, and R. Phippen. The Enterprise Service Bus: Making Service-Oriented Architecture Real. *IBM Systems Journal*, 44(4):781–797, 2005.

[18] A. Slominski, A. di Costanzo, D. Gannon, and D. Caromel. Asynchronous Peer-to-Peer Web Services and Firewalls. In *Proceedings of the 7th International Workshop on Java for Parallel and Distributed Programming*, 2005.

[19] M. Stonebraker. Data services. *Database Engineering Bulletin*, 9(1):4–9, 1986.

[20] Michael Stonebraker. The Case for Shared Nothing. *Database Engineering Bulletin*, 9(1):4–9, 1986.