Asynchronous Peer-to-Peer Web Services and Firewalls

Denis Caromel, Alexandre di Costanzo
INRIA Sophia Antipolis, CNRS - I3S - UNSA, France
First.LastName@sophia.inria.fr

Dennis Gannon, Aleksander Slominski
Computer Science Department, Indiana University, USA
gannon, aslom @cs.indiana.edu

Abstract

In this paper we test the suitability of Java to implement a scalable Web Service that solves a set of problems related to peer-to-peer interactions between Web Services that are behind firewalls or not generally accessible. In particular we describe how to enable reliable and long running conversations through firewalls between Web Service peers that have no accessible network endpoints.

Our solution is to implement in Java a Web Services Dispatcher (WSD) that is an intermediary service that forwards messages and can facilitate message exchanges by supporting SOAP RPC over HTTP and WS-Addressing for asynchronous messaging. We describe how Web Service clients that have no network endpoints, such as applets, can become Web Service peers by using an additional message store-and-forward service ("mailbox"). Then we conduct a set of experiments to evaluate performance of Java implementation in realistic Web Service scenarios, involving intercontinental tests between France and the US.

1 Introduction and Motivation

The emerging trend in Web Services (WS) is to avoid tightly coupled RPC interactions in favor of loosely coupled asynchronous messaging. Initially, SOAP was viewed by many as a better remote procedure call (RPC) [8] mechanism which worked on Internet scale and was capable of passing through firewalls. This view has changed in recent years as the final SOAP specification has focused on messaging and with RPC no longer required. New WS specifications such as WS-Addressing, WS-ReliableMessaging, or WS-Transaction, indicate clearly that asynchronous, long lasting, peer-to-peer interactions (sometimes called conversations) are important to future of Web Services.

Today Internet is built on top of the TCP/IP protocol that is inherently peer-to-peer (p2p). However, the limited supply of IPv4 addresses and, more importantly, use of firewalls and Network Address Translation Systems (NATs) makes it hard to support p2p communication directly on top of TCP/IP. There are a number of solutions proposed including more widespread use of IPv6 and better practices for firewalls that use the extended IPv6 address space and avoid the use of NATs. However the move to IPv6 is not going to replace IPv4 overnight. Instead, many alternative ad-hoc solutions have been proposed. Even through limited, they somehow work with current Internet infrastructure. In particular, file sharing and instant messaging networks proved that such immediate solutions are not only possible but good enough.

In this paper we build on this experience to identify a set of common problems that appear when trying to do peer-to-peer interactions with Web Services that are behind firewalls or not generally accessible. We propose a Web Services Dispatcher (WSD) as a service that is capable of providing reliable and secure peer-to-peer interactions between Web Services peers and can additionally provide load balancing, single sign-on, and service location transparency. Then we examine how Java can be used to implement such solution and evaluate its performance.

2 Related work

There are two separate modes of interactions with our WS-Dispatcher. If the WSD is used with SOAP-RPC then interactions follow a common HTTP Proxy pattern: the incoming HTTP request is forwarded to destination Web Services (after any necessary security or validity checks) and the HTTP response is sent back using the same connection. However this approach is not suitable for a WS that may need a non-trivial amount of time to produce the response. In this cases a TCP connection may timeout before response is available to send back. A significant amount of work which already exists examines performance of HTTP

Proxy servers [17], load balancing, and other optimizations for advanced Web Servers at the TCP level [13]. We have made our WS-Dispatcher implementation modular so that it can be adapted to work in any servlet container within existing commercial products and easily integrated in existing infrastructure. A promising research direction is to generalize the notion of HTTP proxy to further increase performance and scalability [11]. However, in our case we limit ourselves to solutions that are compatible with current WS standards, such as SOAP RPC and messaging. The only other standard beside SOAP that we use is WS-Addressing [10] to allow message level routing.

Web Services are defined in terms of SOAP [9] message exchange patterns. In particular the SOAP processing model allows one to use intermediaries that help with routing of SOAP messages. Technically our WS-Dispatcher is similar to a SOAP intermediary but it is designed to be a transparent service.

In general it is easy to create a very simple dispatcher-like functionality [15], however providing a fully transparent intermediary requires considerable effort. There is already a significant commercial interest in building scalable WS routers or gateways. Consequently there are many companies (such as IBM, BEA, Sonic Software) working on similar products and message routing is a very important part of future commercial web services (including those called "Enterprise Service Bus"). The IBM Web Service Gateway is a typical example of such a product [16]. Gateway is part of the WebSphere Application Server Network Deployment Version 5 [5]. Gateway has an interesting design based around modified open source Web Service Invocation Framework (WSIF [12][4]) which is Apache open source Java project and is designed to allow multiple protocols use when accessing services hosted in Gateway. Our WSD currently only supports SOAP/XML messages but extensions to other protocols, such as binary XML, may be an interesting topic to investigate in future work.

3 Connecting RPC and Messaged oriented Web Service peers

Before we start a detailed discussion of the WS-Dispatcher design we should consider its role in translating semantics between RPC and message oriented Web Service peers. There are few possible choices: a peer acting as a client may make an RPC call or send a message to the WSD that then forwards it to an actual Web Service peer that may be implemented using RPC or message style middleware.

Table 1 describes the matrix of scenarios that must be considered. This relatively simple table becomes more complicated when we consider that both client and service may be locate behind firewall that allows only outgoing connections.

When the client is RPC-based it can use an HTTP connection to receive a response. However this capability is limited by the duration of the TCP connection prior to its time-out. There are clever workarounds that will try to keep HTTP/TCP connection alive and/or reinitiate connection if it fails, but these solutions place a big burden on the client.

However even for an RPC client, if a Web Service peer is behind firewall it is not possible to communicate with it. In case of SOAP-RPC a standard HTTP proxy may be used but a standard HTTP proxy will not be able to do any inspection of the SOAP traffic. Our WSD (and similar commercial products [5]) can alleviate this problem by forwarding RPC connections. This introduces additional processing time (to establish forwarded connection) and generally will not work well if the message has to pass multiple firewalls or, even worse, when the time to generate the RPC response takes longer than HTTP/TCP timeout. Consequently, message oriented processing looks very attractive for Web Services as it allows interaction between peers that may be connected by any number of intermediaries and transport protocols other than HTTP/TCP. It also allows for many message interaction patters and flexible timeout policies.

WS-Addressing (WSA) [10] is gaining popularity as a specification to describe addressing of WS messages. We have used WSA in the dispatcher to facilitate forwarding of messages and it has worked very well. However neither WSA nor RPC addresses the problem of a client that has no accessible network endpoint but wants to receive asynchronous messages.

We propose the solution to this problem by implementing a mechanism similar to a post office mailbox. A Web Service client with no endpoint creates a mailbox and then uses this mailbox address when it needs to receive messages. When the client is ready, it can check the mailbox service (Post Office) for new messages and download them for processing. We call our implementation of this mechanism WS-MsgBox and describe it in more detail below.

WS-Dispatcher with WS-MsgBox provides a complete solution for Web Service peers that are behind firewall but need to communicate asynchronously by exchanging messages.

4 Design and implementation

At this time, we have implemented two versions of the WS-Dispatcher. The first version forwards RPC interactions and the second handles asynchronous messages based interactions.

4.1 Design

In the RPC case, clients wait for a response from the WS and WS-Dispatcher must maintain one connection with the client and a second one with the WS.

	RPC based service	**Messaging based service**
Peer acting as RPC client	Limited but very popular (RPC connection is forwarded) (1)	Very limited (may not work at all if message reply comes too late) (2)
Peer acting as messaging client	Limited: RPC server is a bottleneck (translation of semantics from messaging to RPC) (3)	Unlimited (This is the best situation as there is no transport time limit on sending response) (4)

Table 1. Possible interactions between Web Service peers using WS-Dispatcher.

In a message based approach there is no need to keep connections open, which is good for scalability. In this case WS-Dispatcher works like a forwarding agent that is accepting and forwarding messages. Furthermore in the Message approach it is possible to add new intermediary message oriented service, such as WS-MsgBox or WS load balancer.

We expect that asynchronous forwarding should scale better and be more robust than RPC forwarding. Additionally, in the Message based approach, after the WSD accepts an incoming message, it can queue it for later delivery and, when it is deemed appropriate, multiple messages can be delivered to a destination over one connection which is more efficient than opening multiple short lived connections.

We implemented 2 versions of the WS-Dispatcher: *RPC-Dispatcher* for RPC forwarding and *MSG-Dispatcher* for asynchronous message based services.

Both dispatchers share a common functionality: registry of services. This is a list of web services that are behind the firewall and are to be made accessible through the dispatcher. Each entry in the service registry describes the ”logical” address used by clients and the permanent addresses where the service is implemented. The role of dispatcher is to translate logical address to known physical locations. Hence this registry of services could be used like a directory or Yellow Pages, possibly as a simple browseable list of WSDL files with metadata. Because creating a real registry of services for registering/updating services is independent from forwarding requests, the registry is an independent module in the WS-Dispatcher.

Accordingly the WS-MsgBox service is also an independent service from the RPC-Dispatcher and MSG-Dispatcher. Clients can directly contact the WS-MsgBox service to get responses from a requested WS without invoking the dispatcher.

The WS-Dispatcher design is illustrated in Figure 1. This Figure also describes the task of processing a request from a client to a WS: (1) the client sends a SOAP message to the WS-Dispatcher with a logical name of a WS, (2) the WS-Dispatcher asks the Registry for the logical service name and the Registry returns the physical address of the requested WS (3). Using the real address, the WS-Dispatcher forwards the message to the WS (4). If needed, the WS sends back a response to the message to the WS-Dispatcher