# 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

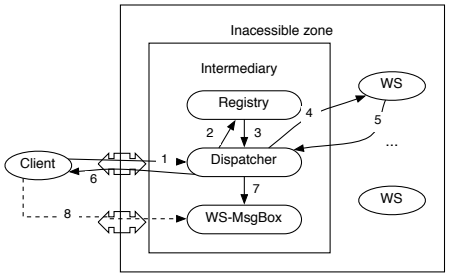


**Figure 1. Design and execution.**

(5). In case of an RPC, call the RPC-Dispatcher directly forwards the response to the Client (6). In the MSG-Dispatcher the response message may be sent directly to the client (6) or the dispatcher may send the message to the WS-MsgBox acting like a post office mailbox (7). Later, the client can retrieve the message response from this mailbox (8).

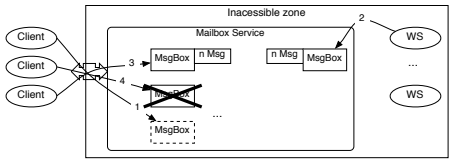


**Figure 2. The WS-MsgBox.**

The WS-MsgBox design is best explained by referring to Figure 2.

The WS-MsgBox provides a set of services for both WS and clients. WS-MsgBox works like a ”real” P.O. mailbox. First clients must create their mailboxes (MsgBox) at the WS-MsgBox (1). These MsgBoxs receive messages from WS or WS-Dispatcher for their owners (2). Clients take messages regularly from their MsgBox (3). When no longer needed the client can destroy its MsgBox (4) to free memory space in the WS-MsgBox service implementation.

All interactions between clients and the WS-MsgBox are RPC, because RPC is typically well supported from a client behind firewalls. The client creates its MsgBox and then sends messages to a WS that is behind MSG-Dispatcher, during the forwarding and waiting response from the WS

the client is free to do something else. When the WS sends a response message to the MSG-Dispatcher, it must then forwards it to sender address. This sender address may be the mailbox address for the client who will pick up message later.

### 4.2 Implementation

The implementation of WS-Dispatcher is integrated in XSUL [6] (WS/XML Services Utility Library). XSUL is a very modular Java library for constructing web services that uses XML. XSUL is derived from a previous work on XSOAP [7] that in turn was derived from SoapRMI.

Our WS-Dispatcher implementation uses the following XSUL modules: HTTP transport (client and server); SOAP 1.1 and 1.2 wrapping/unwrapping; RPC style wrapping; WS-Addressing [10] message manipulation.

Because the RPC and MSG dispatchers must deal with different assumptions about message exchange patterns, their implementations are different too. The MSG-Dispatcher implementation is fully multi-threaded and uses thread pool management, a FIFO queue and the concurrent hash map from the Concurrent Java Library [14] (which is now integrated in Java 1.5). This library is also used for the implementation of the registry service. In contrast, the RPC-Dispatcher is simpler and designed only to process HTTP connections.

The first phase of the implementation consisted of constructing a simple HTTP proxy, called the RPC-Dispatcher, that forwards RPC invocations. It uses one thread to parse the HTTP header, copy the XML message from the request to a new XML document that is then used in the RPC invocation between RPC-Dispatcher and the target WS. After the RPC-Dispatcher receives the result from the WS and copies it to the response for the client and sends it back on the same connection.

RPC-Dispatcher contains a simple registry service that uses text files for mapping logical address with physical address. A concurrent hash map from Concurrent Java Library provides managing concurrent access to the registry service. The same registry code is shared with the MSG-Dispatcher.

The MSG-Dispatcher supplies asynchronous forwarding of WS-Addressing messages between clients and services.

The Figure 3 depicts the multi-threaded implementation of MSG-Dispatcher and illustrates the different synchronization points between threads.

As it is shown in Figure 3 the MSG-Dispatcher manages two pools of threads (the sizes of the pools are configurable). All requests for the MSG-Dispatcher (1) are forwarded to a thread from the first pool of clients' threads (2). These threads are named CxThread, their function is to map logical address with physical address of the WS (3) and parse the WS-Addressing message of the request to modify client's information with MSG-Dispatcher's return address. Next, CxThread passes the modified message to the WsThread (4). WsThread has a First-In-First-Out (FIFO) queue of messages to send and has an open connection for a predefined time with a specified WS. The role of this kind of threads is to send messages directly from their queues to WS (5). Responses from WSs are also treated like requests from clients.

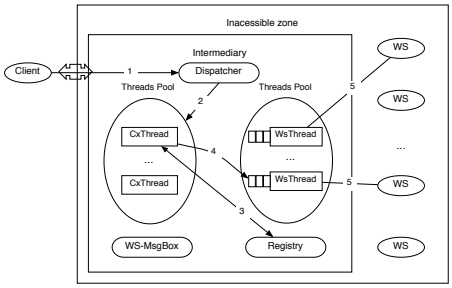


**Figure 3. MSG-Dispatcher implementation.**

With the help of the Concurrent Java Library pools are automatically managed for thread pool operations such as add, pre-create, and destroy.

The P.O. Mailbox (WS-MsgBox) service can be co-located with MSG-Dispatcher or run as a separate service. Clients contact WS-MsgBox service to create a new MsgBox and can use mailbox address when interacting with other WS.

### 4.3 Evaluation

To measure the overhead caused by using the dispatcher and to determine the WS-Dispatcher scalability limits, we conducted experiments between following remote sites: INRIA, Sophia Antipolis, France; the Computer Science Department, Indiana University (IU), US; and a cable modem and a home router in Bloomington, Indiana, US.

All experiments were conducted with a test client that can ramp up number of connections and record statistical data. The test client runs with a specified number of connections (*clients*) and keeps sending echo message (*packets*) for one minute. It returns statistics such as how many calls were made. Essentially it is very similar to the ping command.

We estimate the size of our test SOAP/HTTP message is about 220bytes for HTTP header and 263bytes for the XML message which makes a total of 483bytes (3864bits). This size is the same for both the RPC and asynchronous messages.

The bandwidth to US network endpoints was evaluated with help of a web tool [1] based in California, US. Two different endpoints connectivity speeds (cable internet and backbone internet) were used in the US and one in INRIA

(France) which is a inside institutional network and behind firewall.

- Cable Modem, US (*iuLow*): download 2333 kbps. upload 288 kbps.
- Backbone Internet (Indiana University), USA (*iuHight*): download 3655 kbps. upload 2739 kbps
- INRIA, France:: download 1335 kbps. upload 1262 kbps.

In France two computers were employed: a fast one *inriaFast* (Intel P4@3.4Ghz) and a slow one *inriaSlow* (Intel P3@1Ghz). In US we have used fast one (SunFire 280R 2x1200 MHz) in CS Department and slow one *iuLow* (P3@850Mhz) with cable modem.

#### 4.3.1 RPC Communication

The first experiment evaluates RPC interactions in ”bad” conditions: low bandwidth (or even asymmetrical as it is case for cable modem) and the slowest computers. All interactions results between *iuLow* and *inriaSlow* are plotted in Figure 4. The experiment compared a variable number of clients talking directly to a web service with the same number of clients talking to the WS-Dispatcher in front of the web service. With our ”bad” conditions TCP connection limits are rapidly reached. We measure the number of request that make it to the service (transmitted) as those requests which are lost (not sent). We can see that initially no packets were lost for small number of clients. When number of connections is bigger than 100 we can transfer more messages but we start to loose messages and with 500 connections there is only slight increase in number of messages sent whereas number of lost messages is the same as number of messages delivered. For higher numbers of connections, the improvements in the number of messages delivered are miniscule when compared to the fact that 1000 times more messages are lost than delivered for 2000 connections. This clearly shows that somewhere between 100 and 500 concurrent connections we reached the limit. It seems that using the RPC-Dispatcher had little negative impact on scalability. The next experiments were conducted in ”better” conditions, i.e. enough connectivity *iuHight* and faster workstation *inriaFast* (Figure 5). We had no lost packets at all however we noticed consistent down ”spikes” in performance results and we suspect that they are due to TCP interactions in Java or the OS but we were not able to determine an exact cause. More testing in many different configurations is necessary to answer this question. After 200 connections message throughput does not improve and even gets slightly worsened dues to contention.

#### 4.3.2 Asynchronous Communication

Experiments with asynchronous messages and WS-MsgBox were more difficult to conduct, especially for slow

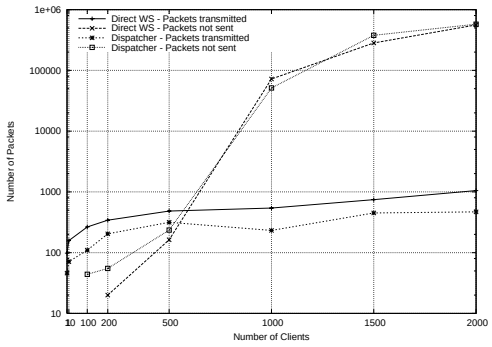


**Figure 4. RPC communication: low broadband.**

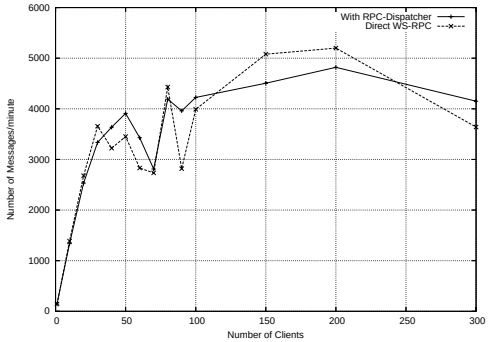


**Figure 5. RPC communication: hight connectivity.**

systems. Figure 6 shows the MSG-Dispatcher results for tests in the ”good” environment. Sending messages directly to the Web Service did not work the best, because the Web Service tried to send back response but the connection was discarded which led to fewer messages accepted by the Web Service. Using the MSG-Dispatcher with WS-MsgBox proved to be the best from performance perspective when the number of concurrent connections is higher than 10. This is because the Web Service had no problem sending response messages to the WS-MsgBox mailbox and there is no delays. When not using WS-MsgBox, the MSG-Dispatcher tried to send a response that was blocked by firewall leading to the slowest performance.

The result of tests for more than 50 clients revealed a very serious bug in the WS-MsgBox implementation. The WSMB was spawning too many threads. For even relatively small numbers of connecting clients (50), if the number of messages sent is high then WS-MsgBox server creates a new thread for each message and each thread tries to send a reply message. Possibly thousands of threads are created

each trying to send a response. That leads to OutOfMemoryExceptions as each thread has local stack allocated in memory and it is known Java limitation and native threads have. We are working to re-design WSMB and we are hopeful to evaluate improved design in future.

This example shows importance of scalability testing: this problem does not manifest itself for small number of connected clients/connections which is typical for infrequently used web services. Therefore if a web service becomes popular but was not tested for scalability users may start to experience undeterministic and very puzzling errors and exceptions and it may take very long time to discover what is causing them.

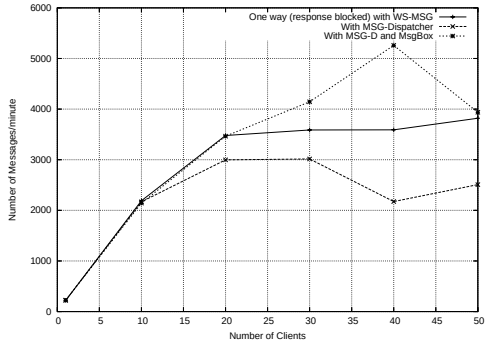


**Figure 6. Asynchronous communication.**

### 4.4 Conclusions and Future Work

The goal of the WS-Dispatcher project was to provide a complete firewall for Web Services with specialized functions like P.O Mailbox, message security inspection, and Registry service. This paper reports on the design and the performance of the system in scalability studies that spanned trans-Atlantic Internet links. It is shown that the use of the system does not degrade service, and in some cases allows for improved performance for message-based traffic.

In the future we plan to integrate a load-balancing system into the Registry service that uses a farm of WS-Dispatchers. To improve performances of this service we would like to integrate a relational database such as MySQL. We would like also to improve Registry service to allow interactive browsing of WSDL files describing services provided by WS-Dispatcher and to allow simple interactions such as checking if service is alive. We are also planning to investigate how WSD can provide authentication and authorization (single sign-on) for web services that do not need to implement security instead relies on WSD to do checks (and load balancing).

We would like also to improve forwarding service by adding hold/retry on delivery to simple one way messaging (HTTP) with messages stored in DB with expiration time. This work would be related with use of WS-ReliableMessaging [2].

We also plan to add security to WS-MsgBox: currently the message box has unique hard to guess address but that is the only protection.

We would like also to integrate better WS-Dispatcher with Grid portals (such as OGCE [3]) to simplify access to services managed in the WSD. When complete, the WS-Dispatcher will be released as open source on http://www.extreme.indiana.edu.

## References


[1] Broadband test. http://www.broadbandreports.com/stest.

[2] Oasis web services reliable messaging (wsrm) tc. http://www.oasis-open.org/.

[3] Ogce open grid computing environments collaboratory. http://www.ogce.org/.

[4] Web services invocation framework, apache software foundation. http://ws.apache.org/wsif/.

[5] Websphere application server network deployment version 5. http://www-128.ibm.com/developerworks/websphere/.

[6] Ws/xsul: Web services/xml services utility library. http://www.extreme.indiana.edu/xgws/xsul/.

[7] Xsoap toolkit. http://www.extreme.indiana.edu/xgws/xsoap/.

[8] Sun microsystems inc. rpc: Remote procedure call protocol. Technical report, In Tech. Rept. DARPA-Internet RFC 1057, SUN Microsystems, Inc., June 1998.

[9] W3c proposed recommendation. soap version 1.2 part 1: Messaging framework, May 2003. http://www.w3.org/.

[10] W3c member submission. web services addressing (ws-addressing), August 2004. http://www.w3.org/Submission/ws-addressing/.

[11] S. M. C. Brooks, M. Mazer and J. Miller. Application-specific proxy servers as http stream transducers. *World Wide Web Journal*, pages 539–548, December 1995.

[12] M. J. Duftler, N. K. Mukhi, A. Slominski, and S. Weerawarana. Web services invocation framework (wsif). OOPSLA Workshop on Object Oriented Web Services, October 2001.

[13] G. D. H. Hunt, G. S. Goldszmidt, R. P. King, and R. Mukherjee. Network dispatcher: a connection router for scalable internet services. *Comput. Netw. ISDN Syst.*, 30(1-7):347–357, 1998.

[14] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[15] X. Liu. Very simple .net web service dispatcher. http://wwww.codeproject.com/.

[16] R. K. N. Mukhi and P. Fremantle. Multi-protocol web services for enterprises and the grid. In *EuroWeb 2002*, Oxford, UK, December 2002.

[17] A. Rousskov and V. Soloviev. A performance study of the squid proxy on HTTP/1.0. *World Wide Web*, 2(1-2):47–67, 1999.