
2007

**Byzantine Fault-Tolerant Web Services for n-Tier and Service Oriented Architectures**

Authors: Sajeeva L. Pallemulle and Kenneth J. Goldman

Web Services that provide mission-critical functionality must be replicated to guarantee correct execution and high availability in spite of arbitrary (Byzantine) faults. Existing approaches for Byzantine fault-tolerant execution of Web Services are inadequate to guarantee correct execution due to several major limitations. Some approaches do not support interoperability between replicated Web Services. Other approaches do not provide fault isolation guarantees that are strong enough to prevent cascading failures across organizational and application boundaries. Moreover, existing approaches place impractical limitations on application development by not supporting long-running active threads of computation, fully asynchronous communication, and access to host specific information. We present Perpetual-WS, middleware that supports interaction between replicated Web Services while providing strict fault isolation guarantees. Perpetual-WS supports both synchronous and asynchronous message passing and enables an application model that supports long-running active threads of computation. We present an implementation based on Axis2 and performance evaluations demonstrating only a moderate decrease in throughput due to replication.

... Read complete abstract on page 2.

Follow this and additional works at: [http://openscholarship.wustl.edu/cse_research](http://openscholarship.wustl.edu/cse_research)

Part of the [Computer Engineering Commons](http://openscholarship.wustl.edu/cse_research), and the [Computer Sciences Commons](http://openscholarship.wustl.edu/cse_research)

**Recommended Citation**


[http://openscholarship.wustl.edu/cse_research/153](http://openscholarship.wustl.edu/cse_research/153)
Complete Abstract:

Web Services that provide mission-critical functionality must be replicated to guarantee correct execution and high availability in spite of arbitrary (Byzantine) faults. Existing approaches for Byzantine fault-tolerant execution of Web Services are inadequate to guarantee correct execution due to several major limitations. Some approaches do not support interoperability between replicated Web Services. Other approaches do not provide fault isolation guarantees that are strong enough to prevent cascading failures across organizational and application boundaries. Moreover, existing approaches place impractical limitations on application development by not supporting long-running active threads of computation, fully asynchronous communication, and access to host specific information. We present Perpetual-WS, middleware that supports interaction between replicated Web Services while providing strict fault isolation guarantees. Perpetual-WS supports both synchronous and asynchronous message passing and enables an application model that supports long-running active threads of computation. We present an implementation based on Axis2 and performance evaluations demonstrating only a moderate decrease in throughput due to replication.
Byzantine Fault-Tolerant Web Services for n-Tier and Service Oriented Architectures

Authors: Sajeeva L. Pallemulle and Kenneth J. Goldman

Corresponding Author: kjg@cse.wustl.edu
Web Page: http://dsys.cse.wustl.edu/

Abstract: Web Services that provide mission-critical functionality must be replicated to guarantee correct execution and high availability in spite of arbitrary (Byzantine) faults. Existing approaches for Byzantine fault-tolerant execution of Web Services are inadequate to guarantee correct execution due to several major limitations. Some approaches do not support interoperability between replicated Web Services. Other approaches do not provide fault isolation guarantees that are strong enough to prevent cascading failures across organizational and application boundaries. Moreover, existing approaches place impractical limitations on application development by not supporting long-running active threads of computation, fully asynchronous communication, and access to host specific information.

We present Perpetual-WS, middleware that supports interaction between replicated Web Services while providing strict fault isolation guarantees. Perpetual-WS supports both synchronous and asynchronous message passing and enables an application model that supports long-running active threads of computation. We present an implementation based on Axis2 and performance evaluations demonstrating only a moderate decrease in throughput due to replication.

Type of Report: Other
Abstract

Web Services that provide mission-critical functionality must be replicated to guarantee correct execution and high availability in spite of arbitrary (Byzantine) faults. Existing approaches for Byzantine fault-tolerant execution of Web Services are inadequate to guarantee correct execution due to several major limitations. Some approaches do not support interoperability between replicated Web Services. Other approaches do not provide fault isolation guarantees that are strong enough to prevent cascading failures across organizational and application boundaries. Moreover, existing approaches place impractical limitations on application development by not supporting long-running active threads of computation, fully asynchronous communication, and access to host specific information.

We present Perpetual-WS, middleware that supports interaction between replicated Web Services while providing strict fault isolation guarantees. Perpetual-WS supports both synchronous and asynchronous message passing and enables an application model that supports long-running active threads of computation. We present an implementation based on Axis2 and performance evaluations demonstrating only a moderate decrease in throughput due to replication.
Byzantine Fault-Tolerant Web Services for n-Tier and Service Oriented Architectures

Sajeeva L. Pallemulle  Kenneth J. Goldman

Department of Computer Science and Engineering
Washington University in St. Louis, St. Louis, MO 63130 USA
{sajeeva, kkg}@cse.wustl.edu

ABSTRACT

Web Services that provide mission-critical functionality must be replicated to guarantee correct execution and high availability in spite of arbitrary (Byzantine) faults. Existing approaches for Byzantine fault-tolerant execution of Web Services are inadequate to guarantee correct execution due to several major limitations. Some approaches do not support interoperability between replicated Web Services. Other approaches do not provide fault isolation guarantees that are strong enough to prevent cascading failures across organizational and application boundaries. Moreover, existing approaches place impractical limitations on application development by not supporting long-running active threads of computation, fully asynchronous communication, and access to host specific information.

We present Perpetual-WS, middleware that supports interaction between replicated Web Services while providing strict fault isolation guarantees. Perpetual-WS supports both synchronous and asynchronous message passing and enables an application model that supports long-running active threads of computation. We present an implementation based on Axis2 and performance evaluations demonstrating only a moderate decrease in throughput due to replication.

KEY WORDS
Web Services, Fault tolerance, Byzantine agreement, n-Tier systems, Axis2, Asynchronous communication

1 Introduction

Enterprises and institutions increasingly use Web Services [1] to provide a wide variety of utilities, ranging from simple mapping utilities (Google Maps [2]) to mission-critical utilities such as payment authorization portals for credit card transactions (Mastercard [3]). Combining functionality offered by multiple Web Services from different providers to perform high-level tasks has become the de facto model for building complex enterprise applications.

Mission-critical Web Services must guarantee correct execution and availability in spite of failures. Fail-stop failures, such as host crashes, can be masked by failing over to other hosts, but achieving Byzantine Fault Tolerance (BFT) [4] requires a higher degree of replication\(^1\) since failures may be caused by malicious attacks and arbitrary software errors in addition to host crashes and network disruptions.

Replicated Web Services that serve multiple applications must ensure fault isolation between applications. To provide such a guarantee, a Web Service must be able to maintain safety (consistent replica state) and liveness (eventual execution of requested tasks) even when interacting with potentially compromised\(^2\) Web Services. Consequently, an execution environment that (1) enables interaction between replicated Web Services with different degrees of replication while (2) guaranteeing safety and liveness is desirable for deployment of mission-critical Web Services.

Existing approaches [5, 6, 7] to building Byzantine fault-tolerant Web Services have failed to gain traction due to several major limitations. Most approaches only allow replicated Web Services to be accessed by unreplicated Web Service endpoints\(^3\). In addition, existing approaches do not guarantee the safety or liveness of replicated calling (client) Web Services if target (server) Web Services are compromised. New directions in Web Services technology such as orchestration (Section 2.2) require support for Web Services with long-running active threads of computation. However, existing approaches only support Web Services that perform short-lived passive computations. Fully asynchronous messaging allows calling Web Services to issue requests in parallel and target Web Services to start processing incoming requests before finishing previous computations. Such a programming model, which is increasingly popular among Web Service developers, is incompatible with existing approaches. Moreover, these approaches enforce determinism in Web Service applications by precluding access to host specific information (e.g., local clock). Section 3 describes limitations of existing approaches in greater detail.

\(^1\)\(^{3f+1}\) state machine replicas are needed to tolerate \(f\) Byzantine faults.

\(^2\)A compromised Web Service has more than \(f\) faulty replicas.

\(^3\)Endpoints may be other Web Services or client applications.
We address these concerns with Perpetual-WS, middleware that augments the Apache Axis2 [8] Web Service execution environment with a Byzantine fault-tolerant transport module and an API suitable for fully asynchronous communication. Perpetual-WS supports Web Services that may utilize a long-running active thread of computation, local clock queries, pseudo-random numbers, and timestamps. The transport module uses Perpetual [9], an algorithm that enables interaction between state machine replica groups while preserving safety and liveness. Our benchmark evaluations show that Perpetual-WS scales well to large replica groups and incurs only a modest decrease in throughput when used to replicate Web Services that perform non-trivial computation tasks.

The rest of this paper is organized as follows. Section 2 presents an overview of Perpetual, Web Services, and Axis2. Section 3 describes unique properties of Perpetual-WS in the context of related work. We present the Perpetual-WS programming model in Section 4. The architecture and implementation details are discussed in Section 5. Section 6 presents macro and micro benchmark evaluations of our implementation, and we conclude in Section 7 with a discussion of future work.

2 Background

In this section we provide overviews of Perpetual, Web Services, and Axis2.

2.1 Perpetual

Perpetual enables two replicated deterministic services to interact using synchronous or asynchronous message exchange models. The safety and liveness of correct services are guaranteed even during interactions with potentially compromised services. For correctness, the Perpetual algorithm assumes that cryptography cannot be subverted and that message delays do not grow faster than time.

2.1.1 Algorithm

We describe the Perpetual algorithm in terms of a target service \( t \), comprised of \( n_t = 3f_t + 1 \) replicas \( t_1, \ldots, t_{n_t} \), and a calling service \( c \) comprised of \( n_c = 3f_c + 1 \) replicas \( c_1, \ldots, c_{n_c} \), where \( f_t \) and \( f_c \) are the upper bounds on the number of faulty replicas tolerated by the target and calling services, respectively.

Each replica \( i \) (target or calling) is composed of a voter \( v_i \) and a driver \( d_i \). The voters and the drivers form two distinct replica groups with the voter and driver of a particular replica co-existing on a single host. Voters of a particular service \( s \) use the Castro and Liskov Byzantine fault-tolerance (CLBFT) [10] algorithm to run agreement on replies to requests originated by \( s \) as well as external requests sent to \( s \) by other services.

Each driver contains an executor, a black box capturing application behavior. Executors model deterministic applications that: (1) request operations on target services and process their replies and (2) execute operations requested by calling services and sending back replies. The requests may be synchronous or asynchronous.

We illustrate the algorithm in Figure 1 by tracing the execution of a request in the non-faulty case. When the executor at calling replica \( c_j \) requests an operation to be performed by \( t \), the driver \( d_j \) sends the request to the voter primary of \( t \) (Stage 1). The voter primary of \( t \) waits for at least \( f_c + 1 \) matching requests before starting CLBFT to
agree on the request (Stage 2). Upon agreement, each voter $v_k$ of $t$ passes the request to its co-located driver $d_k$ (Stage 3) using the local event queue. The executor at $d_k$ dequeues the request, executes it, and sends the result back to voter $v_k$ via driver $d_k$ (Stage 4). Note that the executor at $d_k$ is not required to finish executing a request before starting the execution of the next request. The executor at $d_k$, for example, may deterministically choose to start the execution of the next request while waiting for replies to external requests issued during the execution of the previous request.

To avoid the $n_t n_c$ messages that would result from having all voters of $t$ send replies to all drivers of $c$, each voter of $t$ forwards its reply to a particular voter of $t$, known as the responder (Stage 5). The responder, specified in the original request messages from the drivers of $c$, collects $f_t + 1$ matching replies and forwards the reply bundle (including all authenticators) to each driver of $c$ (Stage 6). When a driver $d_j$ of calling replica $c_j$ receives this message, it authenticates the reply bundle and forwards the result to the primary of $c$’s voter group (Stage 7) that uses CLBFT to agree on the reply (Stage 8). Once agreement has been reached, each voter of $c$ enqueues the result in the local event queue (Stage 9). When an executor of $c$ deterministically decides to consume the result of a request, it pulls that result from the event queue, blocking if necessary until a result for that request is available.

Our previous work [9, 11] describes fault handling, checkpoint generation, and garbage collection in Perpetual.

2.1.2 Implementation

A high quality Java-based prototype of Perpetual has been developed by our group [9]. The implementation contains three main modules. The first module implements CLBFT. The second module implements the core Perpetual algorithm. Both CLBFT and Perpetual Core modules abstract away transport, authentication, and encryption details, which are provided by a ChannelAdapter module. The ChannelAdapter itself achieves transport independence by encapsulating the transport oriented details within Connection modules. The current implementation provides a Connection module that supports SSL/TCP communication.

The implementation uses Message Authentication Codes (MAC) [12] to authenticate all communication. We utilize non-blocking sockets, thread-pooling, and memory-mapped buffers for improved performance.

2.2 Web Services

Web Services are based on a two-tier model, in which, a caller (or client) sends a SOAP [13] message to a target Web Service and expects a reply. However, in reality, the target Web Service may need to contact other Web Services in order to process the original message. For example, when an end-user makes a credit card transaction at an online store, the store Web Service must contact a payment gateway which will in turn contact a bank before authorizing the purchase. Consequently, complex distributed applications may span multiple Web Service tiers that are located across organizational and geographical boundaries.

Large enterprise applications are increasingly built by grouping Web Services using a Service Oriented Architecture (SOA) [14]. Unlike in tiered Web Services applications, where calls to one tier are embedded within Web Services of another tier, Web Services in SOA applications typically provide unassociated sets of functionality. The application depends on an orchestrator that actively executes rules that specify the data flow from one Web Service to another to complete overall tasks. Standards such as the Business Process Execution Language (BPEL) and corresponding BPEL engines (e.g., Apache ODE [15]) facilitate the creation and execution of SOA applications.

Most existing Web Services applications use synchronous communication, in which the caller sends a SOAP message to a Web Service and blocks until it receives a reply message. If the target Web Service takes a long time to process the message or is slow to respond, the caller may be blocked needlessly. Although this scenario may be acceptable for client-to-business (C2B) interactions, valuable processing time may be lost if the caller is a Web Service that initiated a business-to-business (B2B) interaction with another Web Service. Consequently, asynchronous communication, in which callers send SOAP messages to target Web Services and continue to execute their business logic while waiting for a response, is becoming increasingly popular. New standards (e.g., WS-Addressing [16]) as well as new message exchange patterns (MEP) (e.g., conversational Web Services [17]) have been developed to facilitate asynchronous communication. Most Web Services middleware (e.g., Axis2) provides API level support for asynchronous communication.

2.3 Apache Axis2

Apache Axis2 [8] is a popular open source implementation of SOAP [13] written in Java. It provides API level support for sending and receiving SOAP messages. The Axis2 architecture consists of loosely coupled modules that encapsulate high-level functions (e.g., Transport).

Axis2 provides a Client API that supports synchronous and asynchronous communication. Messages are passed to the Axis2 Engine through the Client API. The Axis2 Engine contains an OUT-PIPE that holds a series of handlers that augment the message. The OUT-PIPE can be customized by adding extra handlers. Once a message has passed through the OUT-PIPE, it is handed to a TransportSender. Different implementations of TransportSender may use different protocols (e.g., HTTP, HTTPS, SMTP) to send the message to a matching TransportListener at the receiver.

---

4 An implementation of Axis2 in C also exists.
When a message is received by a TransportListener, it is sent to a MessageReceiver though an IN-PIPE in the Axis2 Engine. The IN-PIPE also contains a series of handlers that can be customized. At a server, the MessageReceiver may invoke requested operations and send results back using the OUT-PIPE. At a client, the MessageReceiver may return results to a thread blocked on a synchronous call or invoke a Callback object to complete an asynchronous call.

3 Contributions and Related Work

We build upon Perpetual and Axis2 to develop efficient middleware for BFT execution of Web Services. In this section, we present unique properties of Perpetual-WS in the context of related work. In particular, we compare Perpetual-WS to Thema [5], BFT-WS [6], and SWS [7].

Interaction between replicated Web Services: Both Perpetual-WS and SWS enable interaction between Web Services with different degrees of replication. Thema and BFT-WS support BFT execution of Web Services through replication and enable replicated Web Services to process requests from unreplicated callers. Thema also enables replicated Web Services to issue calls to unreplicated Web Services. However, Thema and BFT-WS do not support interaction between replicated Web Services.

Fault isolation: Replicated Web Services may encounter compromised target Web Services that may not respond or send different messages to individual replicas of the caller. For liveness, the caller may have to deterministically abort the request. For safety, the calling replicas may have to deterministically choose a single result. Perpetual-WS guarantees the safety and liveness of all non-faulty Web Services even when interacting with potentially compromised Web Services. BFT-WS does not support replicated callers, and neither Thema nor SWS guarantee safety or liveness if target Web Services are faulty.

Long-running threads of computation: In all four approaches the Web Service being replicated must be deterministic. However, Thema, BFT-WS, and SWS all require that Web services be passive, meaning that the state of the application changes only in response to external messages.

Perpetual-WS only requires that the application be single threaded, meaning that active processes such as orchestration can take place within a replicated Web Service application in addition to processing external messages.

Asynchronous communication: Thema and SWS only support synchronous message exchange patterns. Consequently, Web Services that use Thema or SWS may be blocked waiting for responses instead of processing incoming messages. BFT-WS does not support replicated callers, and it does not support asynchronous processing of incoming messages. In contrast, Perpetual-WS allows replicated Web Services to send and receive messages asynchronously. When acting as caller, a Web Service may send a message and continue other processing tasks until it receives a response; when acting as a target, it may start processing incoming requests before completing previous requests (e.g., if a previous request resulted in an out-call).

Host-specific information: Web Service developers may need to access host specific information such as local clock values, timestamps, and random numbers. Thema, BFT-WS, and SWS enforce determinism by precluding local access to such information. Perpetual-WS does require deterministic execution, but takes a more flexible approach by ensuring that function calls to access such information return consistent values on all replicas. (See Section 4.2).

Cryptographic overhead: Thema and Perpetual-WS use message authentication codes (MAC) to authenticate messages while SWS and BFT-WS use digital signatures [18]. MAC calculations are three orders of magnitude faster than digital signature calculations. Consequently, Thema and Perpetual-WS scale better to larger replica groups, which require more messages per request-reply cycle.

Transport independence: The goal of BFT-WS is to achieve maximum interoperability by integrating within the Axis2 handler chain. This model allows BFT-WS to directly use different Axis2 transport modules (e.g., HTTP, SMTP). Perpetual-WS is also transport independent since Connection objects that support different low-level transport protocols can be plugged into the ChannelAdapter of Perpetual (See Section 5). In contrast, Thema utilizes the BASE [19] implementation of CLBFT, which uses a tightly coupled rudimentary UDP based messaging protocol. SWS uses SOAP, but implementation details are not available.

Support for existing Web Services: Thema, BFT-WS, SWS, and Perpetual-WS can all replicate existing passive deterministic Web Services that use synchronous communication without modification to the application code. In addition, deterministic Web Services that have a long-running thread of computation and asynchronous communication can also be replicated using Perpetual-WS with minimal modifications to the to the application code.
Dynamic discovery: Web Services depend on service brokers to resolve endpoint references to actual destination hosts using the UDDI [20] protocol. However, UDDI does not support replicated Web Services. SWS addresses this problem by making simple modifications to UDDI to store and serve information related to each replica host of a Web Service. Thema, BFT-WS, and Perpetual-WS do not currently support dynamic resolution of endpoint references.

In addition to BFT-WS and Thema, many other approaches to tolerate fail-stop failures in Web Services also exist [21, 22, 23, 24, 25]. There has also been some work in supporting replicated-to-replicated interactions [26, 27, 28] in contexts other than Web Services.

4 Perpetual-WS Programming Model

The Perpetual algorithm presented in Section 2.1 only supports BFT execution of deterministic Web Services. Passive deterministic Axis2 Web Services that only use synchronous messaging can be executed within Perpetual-WS without modification. However, Axis2 uses multiple helper threads to support asynchronous messaging and does not guarantee deterministic thread scheduling. Changing the behavior of non-deterministic software to be deterministic in a manner that is transparent to the software is non-trivial and beyond the scope of this paper. Instead, we currently support the following simplified programming model.

4.1 Application Model

Perpetual-WS supports applications implemented using a single ongoing thread of computation. We do not distinguish between server and client behavior. Instead, applications deployed in Perpetual-WS may (1) issue requests, (2) query for incoming requests, (3) query for incoming replies, and (4) issue replies. The Perpetual-WS API provides the tools required for this programming model.

4.2 Perpetual-WS API

The Perpetual-WS API shown in Figure 3 consists of two parts. The MessageHandler API is a natural successor to the Axis2 client API. It provides accessors to obtain incoming requests and replies. The Utils API provides methods that may be used to obtain current time, timestamps, and random numbers. The underlying implementation of these methods guarantees that the return values are consistent across all replicas, independent of the host that executes the software.

A caller may asynchronously send requests to Web Services using the non-blocking send() method of the MessageHandler interface. The caller is required to provide the payload and destination information encapsulated within a org.apache.axis2.context.MessageContext object. The construction of the MessageContext must follow the same rules as when sending a message using the Axis2 OperationClient API.

A caller may request the next available result for any pending request using the getNextResult() method. The method returns the next available reply from the incoming queue, blocking, if necessary, until some reply is available. The receiveReply(MessageContext request) method allows the caller to block until the reply to a specific request arrives. The sendReceive() method enables synchronous invocations on target Web Services.

A target that accepts incoming requests may do so by calling the receiveRequest() method. Once the request has been processed, the Web Service can use the sendReply() method to send the result back to the caller.

A caller may wish to abort requests issued to unresponsive target Web Services. The default behavior in Perpetual-WS is not to abort any outstanding requests. To abort a request, the caller must specify a timeout period using the setTimeoutInMilliseconds() method of the Options object encapsulated within the MessageContext of the request. Although individual replicas may timeout at different points in time, the Perpetual voter group ensures that requests are aborted deterministically on all calling replicas.

When currentTimeMillis() or timestamp() is called, a request is issued to the Perpetual voter group. The primary of the voter group suggests a value which is then voted upon by all the voters. Since this vote may take an arbitrarily long time, these methods may not meet realtime constraints. Calls to random() will result in the creation of new java.util.Random objects using a seed value agreed upon by all of the Web Service replicas.
5 Perpetual-WS Architecture

Both Perpetual (Section 2.1.2) and Axis2 (Section 2.3) architectures are highly modular. Consequently, we were able to design the Perpetual-WS architecture without modifying the core Axis2 or Perpetual implementations. We now describe the Perpetual-WS architecture by tracing the execution of a request during fault-free execution, as shown in Figure 4. We utilize the WS-Addressing [16] support built into Axis2 to enable asynchronous communication patterns.

5.1 Modular Interaction

The calling logic passes a MessageContext to the MessageHandler to initiate the request in stage (1). The MessageHandler augments the MessageContext by setting the wsa:replyTo field and assigning a unique ID to the wsa:messageID field. The MessageContext is then sent to the Transport layer through the Axis2 OUT-PIPE.

The PerpetualSender implements the Axis2 TransportSender interface. Once the PerpetualSender receives the MessageContext, a new Perpetual request is created using the MessageContext as the payload and the value of the wsa:sendTo field of the MessageContext as the target Web Service ID. To support non-blocking calls, the sending thread must not block waiting for a reply from the Perpetual layer. Hence, the Perpetual request is passed to the Perpetual Core using the non-blocking send method of the Perpetual API [9] in stage (2). The Perpetual Core processes the request and sends it to the ChannelAdapter to start stage (3). The ChannelAdapter passes the request to the Connection module that handles all communication with the target Web Service primary. The Connection module adds the relevant authentication information to the request and sends it out through an encrypted connection.

The corresponding Connection module at the target Web Service primary receives the request, authenticates it, and passes it to the Perpetual Core. The Perpetual Core of the primary replica initiates the agreement protocol in stage (4) by forwarding the request to the other target Web Service replicas. Once agreement has been reached, the Perpetual Core at each target Web Service replica places the request in a FIFO queue that is consumed by the Perpetual-Listener through the Perpetual API in stage (5). An ongoing thread at the PerpetualListener fetches the incoming request, extracts the MessageContext from the request message, and passes it to the Axis2 Engine in stage (6). The Axis2 Engine sends the MessageContext through the IN-PIPE to the MessageHandler, which implements the Axis2 org.apache.axis2.engine.MessageReceiver interface. To support asynchronous processing of incoming messages, the MessageContext is then placed in another FIFO queue. Incoming requests are dequeued by the thread that executes target logic through the Perpetual-WS API.

When the target logic is ready to send a reply, it starts stage (7) by calling the MessageHandler, passing in the MessageContexts of both the reply and the original request. The MessageHandler inspects the wsa:replyTo field of the request MessageContext to determine the destination of the reply and sets the wsa:sendTo field of the reply MessageContext with that value. It also inserts the value of the wsa:messageID from the request MessageContext into a wsa:relatesTo field of the reply MessageContext to be used by the caller to match the reply with the request. The path that the reply message(s) takes until it reaches the MessageHandler(s) at the caller mirrors the path of the request. If the original request was synchronous, the MessageHandler returns the reply MessageContext to the caller thread that is blocked waiting for that reply in stage (12). Otherwise, the MessageContext will be placed a FIFO queue to be fetched by the caller thread through the Perpetual-WS API.

5.2 Deployment

To deploy Perpetual-WS Web Services, we utilized the stand-alone deployment process of Axis2. We modified the axis2.xml deployment descriptor to add the MessageHandler and the custom transport modules into the Axis2 stack. The deployment process for a Web Service mirrors that of Axis2 except we require an additional replicas.xml file. Since Perpetual-WS does not currently provide a mechanism to dynamically resolve EndpointReferences to obtain replica group information, static mappings contained in the replicas.xml file are used instead.

---

6 The timeout value is also passed in, to abort the request if necessary.
we implemented a simple increment method to increment a counter at the target Web Service and return the old value of the counter. To simulate non-zero execution time, we used message digest calculations that approximately took the required length of time to complete. We measured time taken to complete 1000 calls and use the average value over three different runs to calculate each data point.

6.3 Experimental Setup

All of our experiments were performed on a dedicated Washington University testbed [34] made up of 2GHz Opteron machines (with 512 MB of RAM) connected via a Netgear GSM7352S Gigabit Ethernet router (with the ping tool reporting 78µs pairwise RTTs). All machines ran RedHat Desktop 4 (kernel version 2.6.9-42.0.3.EL). All tests used Java Runtime version 1.6.0_03 and the RSA/RC4/MD5 SSL ciphersuite. For the TPC-W benchmark, we used MySQL Sever 5.1 along with the MySQL Connector/J 5.1 JDBC driver and Tomcat 5.5.25.

6.4 Experimental Results

As seen in Figure 6, the effects of replicating the PGE and Bank layers is minimal. Although not shown, we also conducted the same experiments using different implementations of the PGE and Bank Web Services that used synchronous requests instead. Overall, the asynchronous PGE and Bank Web Services performed up to 4% better than the synchronous versions. Since only 5 – 10% of all requests to the Bookstore resulted in calls to the PGE, these gains represent a significant improvement in performance.

Our micro-benchmarks mirror a set of experiments that we performed on the underlying Perpetual implementation [9]. The shapes of the resulting graphs bear a striking resemblance to our original results [9]. The overall conclusion that we can draw from this observation is that the cost of authentication and encryption at the Channel/Adapter layer dwarfs the cost of marshaling and demarshaling XML requests at the Axis2 layer. This observation justifies our decision to use point-to-point MACs instead of third party verifiable digital signatures to authenticate messages.

As seen in Figure 7, the overhead of replication is considerable when only null requests are considered. However, the results show that the decrease in throughput as a percentage of total throughput also diminishes as we add more replicas to make Web Services more robust. This observation argues well for the scalability of Perpetual-WS to larger replica groups.

Figure 8 shows the effect on throughput when incoming requests take non-zero time to process. The overhead relative to the case with no replication is also shown. We can see that as requests take longer to process, the overhead of replication rapidly decreases. For example, consider the case of four replicas in both the caller and target replica groups. The throughput increases from 31% (of the
no replication case) for null operations to 66% when a request takes 6ms (typical database access time) to process. These results justify the cost of Perpetual-WS replication for real-world applications.

Figure 9 shows the gain in throughput achieved by issuing parallel asynchronous requests. With 4, 7, and 10 replicas in both the calling and target Web Services, the throughput increased by as much as 225%, 239%, and 227%, respectively, when asynchronous communication was used. These results validate our efforts to support asynchronous messaging in the context of deterministic applications.

7 Future Work

We plan to develop a deterministic thread scheduler for Perpetual-WS by building upon existing work [35, 36]. This extension will enable Perpetual-WS developers to use the Axis2 Client API and write multi-threaded Web Service applications. We also plan to develop a UDDI service for dynamic Web Service discovery. Ultimately, we plan to extend the capabilities of Perpetual-WS to include execution of BPEL [37] processes using the Apache ODE [15] execution engine.

We thank Haraldur Thorvaldsson for his input in designing Perpetual-WS. We also thank Charlie Wiseman and the rest of the ONL team for their support in using ONL. The authors were supported in part by National Science Foundation grants 0305954, 0618266, and 0722328.

References


