Thema: Byzantine-Fault-Tolerant Middleware for Web-Service Applications*

Michael G. Merideth†, Arun Iyengar‡, Thomas Mikalsen‡, Stefan Tai‡,
Isabelle Rouvellou‡, Priya Narasimhan†

† School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA
{mgm,priya}@cs.cmu.edu

‡ IBM TJ Watson Research Center, Hawthorne, NY, USA
{aruni,tommi,stai,rouvellou}@us.ibm.com

Abstract

Distributed applications composed of collections of Web Services may call for diverse levels of reliability in different parts of the system. Byzantine fault tolerance (BFT) is a general strategy that has recently been shown to be practical for the development of certain classes of survivable, client–server, distributed applications; however, little research has been done on incorporating it into selective parts of multi-tier, distributed applications like Web Services that have heterogeneous reliability requirements. To understand the impacts of combining BFT and Web Services, we have created Thema, a new BFT middleware system that extends the BFT and Web Services technologies to provide a structured way to build Byzantine-fault-tolerant, survivable Web Services that application developers can use like other Web Services. From a reliability perspective, our enhancements are also novel in that they allow Byzantine-fault-tolerant services: (1) to support the multi-tiered requirements of Web Services, and (2) to provide standardized Web Services support for their own clients (through WSDL interfaces and SOAP communication). In this paper we study key architectural implications of combining BFT with Web services and provide a performance evaluation of Thema using the TPC-W benchmark.

1 Introduction

Distributed applications composed of collections of Web Services [18, 25] may call for diverse levels of reliability in different parts of the system. One Web Service may have stringent availability requirements, while it may access other Web Services that do not. This characteristic may be by design, or due to the different Web Services being owned or deployed by different organizations. Reliability, *i.e,* continuous correct service, is of particular importance in a multi-tier system, as failures of remote services and nodes, unforeseen vulnerabilities introduced by updates, or additional challenges such as untrusted client access over the Internet cannot be ignored.

Web Services that must be highly available can be built using replication. Byzantine fault tolerance (BFT) [11] is a long studied high-reliability, replication technique that is designed to protect against arbitrary problems ranging from crash faults to software bugs and security violations. Unlike a system that can tolerate only crash faults, a BFT service can function correctly even if a number of its replicas are compromised by a security violation or begin to act arbitrarily instead of according to specification. While BFT requires a higher degree of replication than techniques that tolerate only crash faults (BFT requires a minimum of $3f + 1$ replicas to tolerate f Byzantine faults), recent research [4] has shown BFT to be practical for the creation of certain classes of client–server distributed applications.

Previous BFT applications have been built with different assumptions from those of modern Web Service applications. In particular, a popular class of BFT applications built using state-machine-replication [17] has typically assumed a client–server model, in which the server processes the client-requests without requiring information from other services. Originally designed to support survivable client–server computing, these systems do not provide support for multi-tiered computing in which a replicated BFT service must act as a client of a second service, which may not be Byzantine-fault-tolerant itself. By contrast, the Web Services application model requires a more general, multi-tiered, distributed system architecture, in which the functionality of a single Web Service application is a function of the aggregate behavior of a variety of individual Web Services that may communicate with each other and that

*Merideth and Narasimhan were partially supported by the Army Research Office through grant number DAAD19-02-1-0389 ("Perpetually Available and Secure Information Systems") to the Center for Computer and Communications Security at Carnegie Mellon University and the Air Force Research Laboratory grant number FA8750-04-01-0238 ("Increasing Intrusion Tolerance Via Scalable Redundancy"). Merideth was also partially supported through an IBM Research Summer Internship.

may have diverse reliability requirements. Therefore, reliability techniques for Web Services must allow for being incorporated into selective parts of multi-tiered applications that have heterogeneous reliability requirements. Because of this, BFT systems in their current form do not directly support being used as general infrastructure to build Web Service applications.

In this paper, we introduce Thema, a new BFT middleware system that provides a structured way to build Byzantine-fault-tolerant, survivable Web Services that fit the application model of standard Web Services. Thema includes a client library (Thema-C2RS) that allows client access to Byzantine-fault-tolerant Web Services, a server library (Thema-RS) that is designed to facilitate the creation of these services, and an external service library (Thema-US) that allows an external Web Service to be accessed safely by a Byzantine-fault-tolerant Web Service. From an application perspective, Thema-C2RS enables standard Web Service client-application code to use stub-code generated from the WSDL interface of the service to access a Byzantine-fault-tolerant Web Service as it would any other Web Service. Similarly, Thema-US allows standard non-replicated Web Services to be accessed safely by Byzantine-fault-tolerant Web Services without changes to the application. Finally, Thema-RS supports the standard technique of service skeleton-generation and implementation of WSDL interfaces, while giving the application control over the Byzantine-fault-tolerant application API necessary to support distributed state management and non-deterministic functions. In all three cases, the Thema libraries manage both the interface between the standard SOAP engines and the BFT libraries, and the underlying complexity of the Byzantine-fault-tolerant protocol itself.

Thema involves extensions of both the BASE [15] BFT system and of current Web Services toolkits; together, these extensions allow Thema to be used to create survivable Web Services. First, Thema extends BASE to allow Byzantine-fault-tolerant services to make requests of other services (replicated or not); challenges in doing so are discussed in Section 4. In such a configuration, the replicated Byzantine-fault-tolerant portion of the system must act as a client of some other service, and the service that is being accessed must treat the replicated service as a single client. Second, Thema extends the gSOAP [7] and Apache Axis [22] toolkits to provide BFT with support for the Web Services programming model by (1) allowing Byzantine-fault-tolerant services to implement WSDL interfaces directly, (2) allowing client application code to access these services as they would standard Web Services, (3) allowing Byzantine-fault-tolerant services to act as Web Service clients, and (4) supporting SOAP communication over a BFT protocol.

The contributions of this paper are as follows. Through an analysis of the design of Thema, we study the architectural implications of combining BFT with Web Services. We use the TPC-W benchmark [21] to provide a performance evaluation of the type of BFT Web-Services that can be created using Thema.

The remainder of this paper is structured as follows. Section 2 presents background information on Byzantine fault tolerance and Web Services. Section 3 presents the assumptions made in the design of Thema. Section 4 highlights the research challenges solved in Thema, while Section 5 discusses the design, and Section 6 discusses the implementation of the Thema system. We present an empirical evaluation of Thema in Section 7. Section 8 compares and contrasts Thema to related work, while Section 9 concludes.

2 Background

BFT and CLBFT/BASE. Byzantine fault tolerance [11] is a long-studied high-reliability, replication technique that can tolerate a wide range of types of faults. Under a Byzantine fault model, a faulty node may act arbitrarily (subject to assumptions stated in Section 3); this is in contrast to the more restrictive crash-fault model, which assumes that a faulty node will simply be unresponsive. Although only $f + 1$ replicas are required to tolerate f Byzantine faults if communication is assumed to be synchronous and replicas can authenticate messages, this synchrony assumption is not practical for environments like the Internet, where there is no a priori known upper-bound on communication latencies. In such environments, in which messages can be made to arrive eventually, *e.g.,* through the use of retransmissions or reliable messaging, $3f + 1$ replicas are required to tolerate f Byzantine faults [6].

Castro–Liskov Practical Byzantine Fault Tolerance (CLBFT) [4] is a BFT state-machine-replication protocol [17] and library implementation that can be used to create client–server Byzantine-fault-tolerant applications. CLBFT has two important characteristics that distinguish it from previous BFT state-machine-replication systems [14, 9]. First, CLBFT does not rely on synchrony assumptions (such as an upper bound on communication latency) for safety (linearizability); because of this, it works in asynchronous environments. Second, CLBFT provides good performance because it does not require the use of public-key cryptography during normal operation.

From the perspective of a client application, its CLBFT library sends the request to the CLBFT service and waits for $f + 1$ identical responses from different replicas in the group. These $f + 1$ identical responses ensure that the result they contain is correct, because at least one non-faulty replica states that the result is correct.

From the perspective of the service, each replica acts independently, but performs the same operations in the same total order. For any operation to execute, at least $2f + 1$ replicas must state that they will execute it. This guaran-

tees that at least $f + 1$ non-faulty replicas will execute the request, so the client is guaranteed at least $f + 1$ identical responses. Note that not all replicas need to execute a request for the group to make progress; out-of-date replicas will be brought up-to-date through automated, inter-replica state-transfer.

CLBFT does not provide a mechanism for a replicated service to access an external service in a consistent fashion. In Section 4, we discuss further the challenge that this presents to using Byzantine fault tolerance in multi-tier Web Service applications.

Web Services. An emerging trend in e-business is the adoption of Web services as a technology to integrate applications in a loosely-coupled, multi-tier architecture. Web services promote open, ubiquitous XML-based standards such as WSDL [25] and SOAP [18]. WSDL is a Web services interface definition language. SOAP is a transport-neutral messaging format. Thema does not assume or require any Web services technology other than WSDL and SOAP, nor does Thema modify the semantics of these Web services standards.

3 Assumptions and System Model

We assume that Web Service applications may be composed of multiple Web Services, and that these services may have diverse reliability requirements. These services may be owned and operated by different organizations, and therefore may not be able to share a common infrastructure or fault model.

For each Byzantine-fault-tolerant service, we consider a distributed asynchronous system, and an inherently unreliable transmission medium. While Thema does not require any synchrony assumptions for safety, Thema does, like BASE, assume that messages are eventually delivered for liveness. Our fault model considers processor- and process-crash faults, communication faults such as message losses and message corruption, and Byzantine/arbitrary faults in processes and processors.

We assume that faulty replicas may behave arbitrarily, but are computationally bound so that they cannot subvert cryptographic techniques (such as message authentication and digital signatures).

We assume that steps are taken to prevent common-mode failures of multiple replicas. This is a standard assumption in Byzantine fault tolerance that can be handled in some cases by n-version redundancy. For some services, multiple versions may already exist; in this case, the BASE library provides support for making use of these different versions together. Even if diverse implementations are not available, and all replicas run the same service code, BFT can still mask arbitrary faults like heisen-bugs; security risks might be alleviated in a practical setting by taking steps such as running the replicas on different platforms or keeping them in different administrative domains.

We assume that non-replicated tiers (clients, external services) can be configured to communicate with the replicated service: clients need an additional client software library provided by Thema, and external services need an external service library, but this does not require modification of the applications. Instead, the libraries provide low-level protocol code that is integrated with standard SOAP tools; in Section 6, we discuss the integration that Thema currently provides of BASE with gSOAP [7] and Apache Axis [22].

We assume that each node has a public–private key-pair and that distribution of the public keys is handled outside of the Thema system. In Thema, as in BASE, clients and services use public-key cryptography to establish symmetric session keys that are used during operation to create less expensive message authentication codes (MACs).

4 Challenges Addressed by Thema

Thema is designed to address the challenges of creating distributed applications composed of multiple Web Services, some of which may be Byzantine-fault-tolerant. This section presents three of these challenges: using BFT in this mixed fault-model; allowing Byzantine-fault-tolerant services to make safe requests of external services, and providing SOAP and WSDL support for Byzantine-fault-tolerant applications.

4.1 Working in a Mixed Fault-Model

For a Byzantine-fault-tolerant Web Service to be used in Web Service applications, it must support access from non-replicated clients and access to non-replicated Web Services. This creates three problems: (1) clients must treat the Byzantine-fault-tolerant Web Service as a single entity, (2) the Byzantine-fault-tolerant Web Service must guarantee internal consistency when acting as a client, and (3) external Web Services must treat a Byzantine-fault-tolerant Web Service acting as a client as a single entity. Previous BFT systems have addressed problem (1), but not problems (2) and (3).

Clients must treat the Byzantine-fault-tolerant service as a single entity. Unlike systems that support only crash-faults, the fact that replicas in a BFT system may return incorrect values means that the client must directly authenticate and verify the responses returned by up to $2f + 1$ replicas (in order to get $f + 1$ identical responses) before choosing a result; to get the benefit of BFT, the client cannot rely on any service that is not Byzantine-fault-tolerant to perform these operations for it. As seen in Section 5, Thema applications do not need to be aware of this complexity because it is managed by the infrastructure.

A Byzantine-fault-tolerant service must guarantee internal consistency when acting as a client. If it accesses a non-Byzantine-fault-tolerant external service, it may choose to

protect itself from the possibility of its non-faulty replicas receiving and using incorrect results. The first step is for replicas to authenticate the responses from external services; this authentication can be done as it is done in BASE, either by having the external service digitally sign the result using public-key cryptography (so that all replicas can verify the result if any non-faulty replica receives it), or by authenticating the result using MACs. However, if a faulty external service were to return different results to different replicas of a Byzantine-fault-tolerant Web Service acting as a client, the state of the replicas could diverge unless the replicas do something to verify that they have all received the same result. This may require an additional agreement phase like the one done for client requests.

Finally, external services must treat a Byzantine-fault-tolerant Web Service acting as a client as a single entity. Because replicas may be faulty and may make requests that were not intended by the application developer, this requirement implies that an external service cannot perform an operation requested by some of the replicas of a Byzantine-fault-tolerant service unless it is certain that enough replicas have requested the same operation. Because the BASE protocol ensures that no non-faulty replica will execute an operation unless at least f other non-faulty replicas will eventually execute that operation, the external service needs to wait until it knows that at least one non-faulty replica has requested the operation. In order to verify this, it must authenticate at least $f + 1$ identical requests from different replicas. As in the case of the client of the Byzantine-fault-tolerant Web Service, the external service middleware also must directly authenticate and verify the request messages to get the benefit of the BFT guarantees of the Byzantine-fault-tolerant Web Service; it cannot rely on any service that is not Byzantine-fault-tolerant to perform this operation.

4.2 The Asynchronous Operation of BASE

The correct operation of BASE hinges on being able to collect messages from groups of servers, as described in Section 2. As detailed in [4], for normal operation of the protocol, each replica must collect at least $2f + 1$ messages from different replicas (including itself) in the different stages of the protocol. Each replica is guaranteed by the fact that there are $3f+1$ replicas in the system, out of which at most f can be faulty, that it will be able to collect at least $2f + 1$ messages eventually. It is important to note that in the case where the maximum number f replicas are faulty, all $2f + 1$ non-faulty replicas will need to send messages; for the system to be live, $2f + 1$ non-faulty replicas must, at all times, at some point in the future be ready to participate in the protocol. It is partly for this reason that naive approaches to adding multi-tier support to BASE can lead to hard-to-detect, incorrect protocol-operation that would only occur in corner-cases not touched by performance testing in the fault-free case.

The asynchronous nature of the protocol means that request execution in BASE can be performed at different replicas at different times. In fact, non-faulty replica X can execute request A in a different view from non-faulty replica Y. The BASE protocol guarantees that if a request executes at any non-faulty replica, the request will be assigned the same sequence number in subsequent views, so request execution across views will not result in any inconsistency between non-faulty replicas.

Assume that the execution of a given request requires access to an external service and that f replicas in the system are faulty. According to the BASE protocol, any replica is free to execute the request once that request has committed locally, *i.e.,* once the replica knows that $2f + 1$ replicas have claimed that the request prepared. Of these $2f + 1$ replicas that claim that the request has prepared, at least $f + 1$ are non-faulty; assume that the other f are indeed the faulty nodes in the system. In general, to avoid potential state-corruption, in BASE, non-faulty replicas cannot execute additional requests before completing the execution of the current request. Suppose that the replica that committed locally sends its request to the external service and blocks, waiting for a response.

Now, as discussed in Section 4.1, the external service accessed by the request should not accept the request until at least $f+1$ replicas have requested it; otherwise, it might perform an operation requested only by faulty replicas. Therefore, upon receiving the request from the replica that committed locally, this service will not return a response until it also receives a corresponding request from at least f other replicas.

The problem here is that the one replica that committed locally now is blocked waiting for a response instead of participating in the BFT protocol. If but a single replica, from the group of $f + 1$ non-faulty replicas that is prepared, requests a view change before committing locally, the view change will never happen (because $2f + 1$ replicas must request the new view, but f are faulty and one non-faulty replica is not participating in the protocol, leaving only $2f$ replicas for the view change), and the external service will never get f additional requests, because one of these f replicas is requesting a view change. The f faulty replicas cannot be relied upon to solve this problem.

Section 6.1.1 presents the solution to this problem that is employed in the current implementation of Thema.

4.3 Supporting SOAP and WSDL

Web Services typically communicate via SOAP messages that are sent over HTTP. This implies a reliable messaging layer (since HTTP runs over TCP/IP). Web Service clients wait for a single reply to their request. Web Services

may communicate securely using authenticated, encrypted HTTPS. BASE clients, on the other hand, communicate via BASE messages that are sent using unreliable UDP, with retransmissions to handle message losses. BASE clients wait for up to $2f+1$ replies (in order to get $f+1$ identical replies) for a single request. BASE provides its own authentication via MACs, but does not encrypt messages.

Web Service toolkits provide tools for Web Service client and service application development that allow for a clean separation of application logic from communication details. WSDL is provided as a standard interface definition language, and stub and skeleton generators allow these interfaces to be called using the standard method invocation procedures of the application programming language. BASE provides a client and service API. The client API provides a way to invoke service operations; how these operations are identified or described is left to the application developer. The service API provides additional functions for state management that are required for stateful BFT services.

As described in Section 6.2, Thema allows clients and services to be programmed as standard Web Service application components by providing an integration of SOAP toolkits and BFT support.

5 Design of Thema

Thema consists of a client library (Thema-C2RS), a BFT service library (Thema-RS), and an external service library (Thema-US). Each of these libraries provides support for the standard multi-tier Web Service programming model and SOAP communication.

5.1 Overall Architecture

Figure 1 represents the architecture and communication paths of a three-tier Thema Web Service application. Each node, be it a client (first tier), a Byzantine-fault-tolerant Web Service (middle tier), or an external Web Service (third tier), has a Thema middleware library for BFT communication, SOAP messaging, and Web Service programming. In Figure 1, the client invokes a request on a Byzantine-fault-tolerant Web Service, which in turns invokes a request on an external Web Service. The sequence of interactions (also labeled in Figure 1) is:

1. The client application calls a stub function, which transparently executes a request for service provided by the Byzantine-fault-tolerant Web Service and blocks, waiting for a reply. The client's request is converted into an XML SOAP request message by the client's SOAP engine. This SOAP request is sent to the Thema-C2RS library, which bundles the request into a BASE request message, and sends it to the primary replica of the Byzantine-fault-tolerant Web Service. The request message then travels across the network

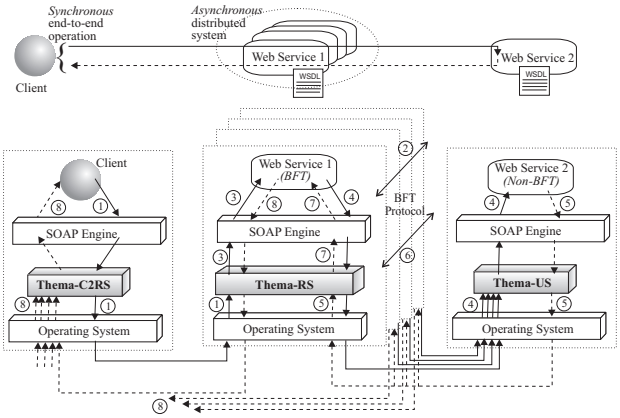

Figure 1. The architecture and communication paths of a three-tier Thema application.

to the Byzantine-fault-tolerant Web Service. When the request message arrives, the Thema-RS library receives it.

2. Before executing the request, the replicas of the service must reach agreement on the ordering of the request. This step uses the BASE protocol, consisting of three phases in normal-case operation, which are described in [15]. Once the replicas have reached agreement, at least $2f + 1$ replicas have agreed to the ordering, so at least $f + 1$ non-faulty replicas will eventually execute the request given the ordering; this is sufficient to ensure that the replicated group stays internally consistent.
3. When the replicas have reached agreement on the request ordering, the request is sent at each replica to the replica's SOAP engine, where it is dispatched to the corresponding Web Service function.
4. The Web Service application logic calls a stub function, which transparently invokes a request on a non-replicated external Web Service and blocks, waiting for a response.[1] As in Step 1, this request is converted into an XML SOAP request message by the replica's SOAP engine. This SOAP request is sent back down to the Thema-RS library, which bundles the request into a BASE request message and sends it to the external Web Service. Because at least $f + 1$ non-faulty replicas will eventually execute the same request, at least $f+1$ identical request messages will be sent by different replicas to the external Web Service. The Thema-US library of the external service waits to receive $f+1$ identical request messages from different replicas in the group, and sends the request to the SOAP engine, where it is dispatched to the corresponding function.

[1] Note that the BFT protocol logic of the replica must continue to run, as described in Section 4.2, even as the application logic remains blocked waiting for the stub function to return.

5. The result of the external Web Service operation travels back down through the external service's SOAP engine, where it is converted into a SOAP response message, and down to the Thema-US library, which bundles the response into a BASE reply message. Thema-US sends this reply message to all of the replicas in the Byzantine-fault-tolerant Web Service. The Thema-RS library of each non-faulty replica receives a copy of the reply.
6. The replicas may now perform an agreement phase on the reply message to ensure that their state does not diverge as a result of a faulty external service.
7. The SOAP response message is extracted from the BASE reply message and is sent back to the application function that originally made the request.
8. The application returns a result to the original client. The result is first converted into a SOAP response message by the replica's SOAP engine, and is then bundled into a BASE reply message by the Thema-RS library. Thema-RS sends the BASE reply message across the network, back to the client. The client's Thema-C2RS library waits for $f + 1$ identical replies, and then unpackages the SOAP response message and sends it up through the client's SOAP engine. The SOAP engine sends the return value specified in the response message back to the client application code.

6 Implementation

Our current implementation of Thema relies on the BASE library implementation [15] of CLBFT for two-tier BFT support, but involves modifications to BASE to allow BFT replicas to make calls of external services during application execution. In addition, Thema provides support for the Web Services programming model by integrating with the gSOAP [7] and Apache-Axis [22] SOAP toolkits.

6.1 Changes to the BASE Library

Thema's underlying BFT library, BASE, provides a client library and a server library. Thema includes modifications of the BASE server library to allow a service to make requests of external services (Thema-RS), and provides a library for allowing external services to support requests from replicated BFT services (Thema-US). Although the BASE library supports authentication using either digital signatures or MACs, our current implementation of Thema uses MACs for all communication, for improved performance. The Thema client library (Thema-C2RS) is based directly on the BASE client library.

6.1.1 BFT Service Library (Thema-RS)

Our implementation of Thema-RS builds on the BASE server library. Most of the modifications are to allow consistent protocol operation during request execution, as required by Section 4.2. We have disabled the proactive recovery functionality [5] of the BASE library, and the tentative execution optimization, both of which can cause the state of a replica to rollback. When invoking a method on an external service, each replica communicates directly with the external service; the Thema modifications make it safe for the replica to do so. Our current implementation does not perform agreement on the response from an external service, although this extension is possible.

All non-faulty replicas that are waiting for a response from an external service must continue to participate in the BASE protocol, otherwise the system can deadlock, as described in Section 4.2. Because the application code written to run on top of Thema is the code that will initiate a call to an external service, the middleware must either provide a way to run the application logic asynchronously from the protocol logic, or to suspend the application code and resume it when the response arrives.

The solution in our current prototype of Thema involves suspending and resuming execution. The BASE library is single-threaded. Our Thema implementation adds multithreading (through the `pthreads` library) to provide one thread for the BASE protocol, and a second thread for the execution of requests. The Thema-RS library implementation must take care to avoid state corruption, which can occur because both the application and the BASE protocol can update the state of a replica. For example, in BASE, state transfer can be used to synchronize the state of a slow replica with other replicas. If the execution thread modifies state during state-transfer that is initiated by, and running in, the protocol thread, inconsistency can occur.

In our prototype, to provide state consistency, only one thread runs at a time. The protocol thread runs until the time when a request is ready to be executed. At that point, the protocol thread yields to the execution thread, which yields to the protocol thread either when it finishes request execution, or when it sends a request to an external service. When the protocol thread receives a response from an external service, it yields to the execution thread, which continues the application execution from the point of the matching request, using the response. If the state is updated by the protocol thread (for example, because of BASE state-transfer) while the execution thread is waiting for a reply from an external service, the execution thread aborts execution of the current request.

6.1.2 External Service Library (Thema-US)

Our implementation of Thema-US is based on the BASE client and server libraries. Like the BASE client and server libraries, Thema-US communicates using UDP with message authentication. Thema-US waits for $f + 1$ identical request messages from different replicas from the same service before executing a request. Upon the completion of the execution of the request, Thema-US sends the result to all

replicas. Our current implementation of Thema-US maintains a log of all replies that it has sent, so that replicas can re-request a reply message, which may be lost due to the use of UDP. A future implementation of Thema-US might retain replies only until it is certain that the replicated group has enough information to retransmit the reply internally.

6.2 Standard SOAP and WSDL Support

gSOAP Service (C++). The integration with the gSOAP skeleton compiler allows a BFT Web-Service to be implemented in C/C++ based on a WSDL interface that describes its methods. The normal-case operation of the standard gSOAP functionality is to read a SOAP message from a socket, parse the message, and dispatch the request to the appropriate service function. The gSOAP library provides a mechanism to instruct it to use a specific socket for reading and writing.

The Thema integration is as follows. When the Thema service library (Thema-RS or Thema-US) is initialized, it replaces the gSOAP socket with a UNIX domain socket obtained through the `socketpair()` function of the Socket API. When a Thema client-request arrives over the network, the Thema library extracts the embedded SOAP request from the message payload, and writes the contents of the SOAP message to the socket pair. The Thema library then instructs the gSOAP library to read from the socket and to execute the appropriate application method. After the application method has completed, the Thema library reads from the socket pair a SOAP response, which is created by the gSOAP library, bundles this response into a BASE reply message, and sends the message to the client.

The BASE server library provides an API that the application code must use when updating the service state; this API is required so that the replicated service can do efficient state management (checkpointing and garbage collection) and transfer. Our current implementation of Thema provides the same API to the Web Service application.

gSOAP Client (C++). The integration with the gSOAP stub compiler allows for any Web-Service C++ client programmed using the gSOAP toolkit to access a BFT Web-Service without modification of the application logic. The standard gSOAP C/C++ client stub functions, which are generated from the Web Service's WSDL file, work as follows. gSOAP connects a socket to the remote Web Service. The C/C++ client stub function translates the application's function call into a SOAP request message, which it writes to the socket. gSOAP waits for a SOAP reply message, which it reads from the socket, converts to the appropriate C/C++ return values, and returns to the application.

The Thema integration is as follows. gSOAP provides a mechanism to replace the function that it uses to connect to a socket, the function that it uses to write requests to the socket, and the function that it uses to read replies from the socket. Our current implementation replaces these standard gSOAP functions (which operate on TCP sockets) with Thema functions. The standard `connect` function is replaced with one that initializes the BASE client library. The `write` function, which is used to send requests, is replaced with one that fills a BASE request message buffer and sends it. The `read` function, which is used to receive replies, is replaced with one that reads from a BASE reply message buffer. In this way, gSOAP transparently sends SOAP messages over the BFT transport instead of over TCP.

Axis Client (Java). The integration with Apache Axis allows for any Web-Service Java client programmed using the Apache Axis tools to access a BFT Web-Service without modification of the application logic. Our current implementation is as follows. Thema provides a BFT Java transport class (ThemaTransport) and a Java native wrapper around the C++ BASE client library. ThemaTransport can be loaded by an unmodified Axis client library based on the type (BFT or other) of the Web Service. Axis translates Java method calls, made by the application, into SOAP request messages, which it passes to ThemaTransport, and receives SOAP replies, which it translates into Java return values, from ThemaTransport. ThemaTransport makes use of the Java native wrapper around the C++ BASE client library. ThemaTransport bundles the SOAP messages received from Axis into the payload of BASE messages and sends them using the BASE client library. The BASE client library waits for $f + 1$ identical responses, and returns the BASE message to ThemaTransport, which unpackages the SOAP response and passes it to Axis.

7 Performance Evaluation

This section describes our experimental results obtained through the use of a micro-benchmark and the TPC-W benchmark [21]. The micro-benchmark is designed to test the performance of Thema in a three-tier configuration, while the TPC-W benchmark is used to test the impact of making a critical service Byzantine-fault-tolerant.

The micro-benchmark measures the round-trip response time from the client's perspective for a three-tier Web Service application. We compare two versions of the application: a Thema version, in which the middle-tier is a BFT Web Service; and a non-replicated gSOAP version in which the middle-tier is a non-BFT Web Service. Both versions use the identical application logic implemented in C++, but the Thema version uses SOAP communication over BFT (UDP), while the gSOAP version uses SOAP communication over TCP. The client program is written in C++.

We use the Wisconsin TPC-W benchmark implementation [3] as a macro-benchmark in order to measure the impact of making one tier of a multi-tiered application Byzantine-fault-tolerant. The TPC-W scenario is designed to simulate the operation and workload of an online book-

store application. The primary metric is the number of web interactions per minute (WIPS) that the bookstore processes. When an order is submitted by a Remote Browser Emulator (RBE), the specification calls for the credit card information and order identifier to be sent to a remote Payment Gateway Emulator (PGE) for authorization. The PGE computes a random authorization string of 15 characters, records this string along with the order identifier, and returns the authorization string to the bookstore.

For our evaluation, we create two versions of the PGE: a Byzantine-fault-tolerant Thema Web Service version, and a non-replicated gSOAP version. In both cases, we add a non-replicated external Web Service on which the PGE relies for credit-card information. Our PGE acts as specified by the benchmark, but makes a call to the external service as part of its processing. As illustrated in Figure 2, credit-card verification in our modified TPC-W scenario is actually a four-tier process. The RBEs (tier a) place orders with the bookstore (tier b), which causes the bookstore to call the PGE (tier c), which calls the external credit-card service (tier d).

The Wisconsin implementation of the TPC-W benchmark is made up of Java servlets that together perform the bookstore functionality, and Java client programs that act as RBEs, simulating bookstore customers, to provide one of three specified workloads (shopping, browsing, or ordering). The Wisconsin implementation does not provide code for the PGE or simulate a call to the PGE, so we modify the code to call the PGE as specified, using the Thema Java client support (Section 6.2), which transparently chooses the BFT or HTTP communication library based based on the type of the PGE. The benchmark specification requires the call to the PGE to take at least two seconds; we do not follow this requirement, as it would mask the overhead of calling the PGE altogether.

7.1 Experimental Setup

We run our experiments on the Netbed/Emulab [24] testbed. Each machine is an 850 Mhz Pentium III with 512M RAM. Each node runs Redhat 9 kernel 2.4.20. Our experimental setup consists of two configurations, one for the micro-benchmark, and one for the TPC-W benchmark. All C++ code is compiled with debugging information turned off, using GNU g++ 2.95. We use gSOAP version 2.7.0 and Apache Axis version 1.2 RC1.

For the micro-benchmark, we use six machines in the BFT case (four middle-tier replicas, one client, and one external service), or three machines in the non-replicated case (one middle-tier service, one client, and one external service), connected to a 100Mbps LAN. This allows us to test the round-trip latency of the Thema system without the additional overhead of message delays or packet loss.

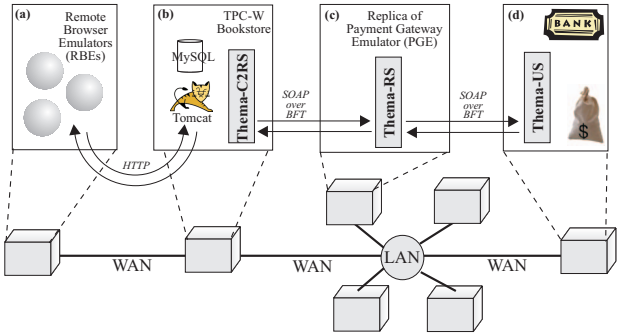

Figure 2. Experimental configuration of the TPC-W benchmark.

For the TPC-W scenario, we use eight machines. One machine runs the RBEs with Java J2SE 5.0. This machine is connected via a 10Mbps link to the TPC-W bookstore machine, which uses the open source servlet container Apache Jakarta Tomcat version 4.1.31 with Java J2SE 5.0 to run the servlets, the open source MySQL database server version 4.1 to store the bookstore information, and the Thema-C2RS library with the Thema Java client to call the PGE. The bookstore acts as a Thema Web services client of the C++ Web Service PGE (which is either a single machine in the non-replicated case, or four machines connected to a 100Mbps LAN, implemented using Thema-RS with the Thema C++ service support, in the replicated case), to which it is connected via a 10Mbps link through a gateway machine. Finally, the credit-card Web service runs on its own machine, is implemented using Thema-US and the Thema C++ service support, and is accessed by the PGE through the gateway and via a 10Mbps link. Both the Thema and gSOAP PGE services, and the credit card services, run identical implementations of the C++ PGE application code.

7.2 Micro-Benchmarks

Figure 3 shows the round-trip times for the three-tier micro-benchmark with varying request sizes. Each data point represents the average latency value for that request size over three independent runs of 1000 requests each. All requests are issued sequentially from a single client machine. For Thema, the latency scales linearly with request-size in the request-size range. For the non-Thema, non-BFT, gSOAP case, the same is true except for the case of the 6K request, which causes a reproducible spike in average latency, due to a number of requests taking more than 200,000 microseconds.

7.3 TPC-W: Macro-Benchmark

Figure 4 shows results obtained from the TPC-W benchmark using varying numbers of RBEs. The PGE authorization request size is approximately 580 bytes, including the soap envelope. All runs were done with the ordering work-

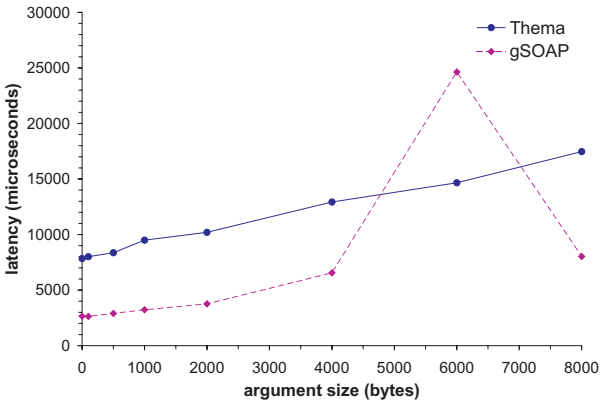

Figure 3. Thema and gSOAP (non-replicated) for varying request sizes on the LAN.

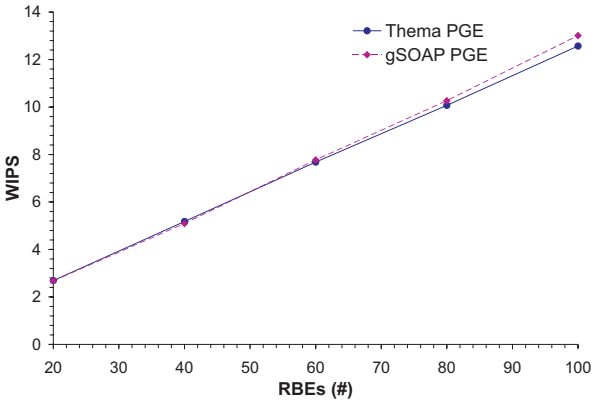

Figure 4. Change in WIPS resulting from varying the number of RBEs.

load, which is the TPC-W workload that most often requires a call to the PGE (approximately one call per every ten interactions). The number of WIPS obtained by the benchmark is very similar for both versions of the PGE.

8 Related Work

Web Service Reliability and Security. Various XML-based standards for Web services reliability and security have been proposed as part of the Web services platform architecture (the set of WS-* specifications) [23]. These specifications define message formats and protocols to support various reliable messaging and security concerns, including guaranteed delivery, message ordering, and authentication, which relate to the topic of this paper. However, the WS-* specifications are focused on interoperability between services that execute in heterogeneous environments. They suggest ways to encode the non-functional properties as part of the SOAP messages. Thema is not primarily concerned with interoperability but addresses reliability and security at the transport layer: Thema sends SOAP messages over the BFT transport. To address non-functional requirements on the transport, messaging, and even application layers, Thema can complement the WS-* specifications. For example, [19] discusses various ways to implement Web services reliable messaging. Additional work is needed in defining ways of reconciling non-functional properties at the various layers.

Birman *et al.* [2] argues that Web Services need to become more autonomic in order to increase reliability; the paper does not address BFT specifically. Townend and Xu [20] seeks to provide higher reliability for Web Services through the dynamic discovery and construction of voting groups of functionally-equivalent Web Services.

Byzantine Fault Tolerance. A great deal of work has been done on Byzantine fault tolerance and making it efficient [12, 9, 4, 10]. One aspect of Thema is the extension of the BASE library [15] to support multi-tier use of BFT. At a high-level, our solution (presented in Section 6.1.1) to the problem presented in Section 4.2 uses a mechanism similar to making the agreement phase of the protocol run asynchronously from the execution phase in a fashion similar to [27, 10], which do so to reduce the number of dedicated execution nodes from $3f + 1$ to $2f + 1$, to allow the insertion of a privacy firewall, and to increase throughput. The implementations of both systems also use BASE, but run separate processes for agreement and execution, instead of multi-threading a single replica process.

In previous work considering multi-tier BFT systems [1, 8, 26], every tier is BFT replicated. Fry and Reiter [8] explores enabling objects implemented as Byzantine-fault-tolerant quorums to act as clients of other objects; the properties of quorum systems are different from those state-machine replication systems like BASE. Two multi-tiered Byzantine-fault-tolerant DNS systems [26, 1] have been presented, which both use the CLBFT protocol; neither system addresses the problems presented in Section 4 directly. Ahmed [1] proposes a communication technique between tiers to reduce message overhead to increase scalability; such a technique may be appropriate for Thema.

One aspect of Thema is the Web Services programming model; other projects have explored and provided techniques for building BFT systems with other programming models. The Immune System [13] provides a CORBA distributed object programming model using the SecureRing [9] Byzantine-fault-tolerant group-communication protocol, which provides different safety and liveness guarantees than CLBFT. The ITDOS project [16] looks to provide a framework for building heterogeneous Byzantine-fault-tolerant distributed object CORBA systems on top of CLBFT, but provides no implementation details. BASE [15] provides a way to use BFT

replication support for pre-built applications. In the original CLBFT library, the application had to be designed to be compatible with BFT; BASE allows the application to be treated as a black box. The important application behavior is distilled into an abstract specification, and the black-box application is wrapped to conform to the specification.

9 Conclusions

This paper has presented Thema, a new middleware system for creating Byzantine-fault-tolerant, survivable Web Services that support the normal Web Services application and programming models. In this paper, we have introduced a number of challenges in combining Byzantine fault tolerance with Web Services, including working in a mixed-fault model, adding multi-tier support to BFT without compromising correctness, and providing SOAP and WSDL support for BFT. We have presented the design and implementation of Thema, and provided a performance analysis using the TPC-W benchmark.

Acknowledgments. We would like to thank Rodrigo Rodrigues for providing the BASE source-code on which we based our implementation of Thema, Miguel Castro for discussions on the workings of the CLBFT protocol, and Mike Dahlin and Ramakrishna Kotla for providing the source code for their High-Throughput BFT implementation. We would like to thank Deepti Srivastava for discussions about Thema, BFT, and the paper, and the anonymous referees for helpful feedback on drafts of the paper.

References

[1] S. Ahmed. A scalable Byzantine fault tolerant secure domain name system. Master's thesis, MIT, Jan. 2001. Also as Technical Report MIT-LCS-TR-849.

[2] K. Birman, R. van Renesse, and W. Vogels. Adding high availability and autonomic behavior to Web services. In *International Conference on Software Engineering*, pages 17–26, May 2004.

[3] H. W. Cain, R. Rajwar, M. Marden, and M. H. Lipasti. An architectural evaluation of Java TPC-W. In *International Symposium on High-Performance Computer Architecture*, page 0229, January 2001.

[4] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Symposium on Operating Systems Design and Implementation*, 1999.

[5] M. Castro and B. Liskov. Proactive recovery in a Byzantine-fault-tolerant system. In *Symposium on Operating Systems Design and Implementation*, 2000.

[6] C. Dwork, N. A. Lynch, and L. J. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[7] R. A. V. Engelen and K. A. Gallivan. The gSOAP toolkit for Web services and peer-to-peer computing networks. In *IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 128, May 2002.

[8] C. P. Fry and M. K. Reiter. Nested objects in a Byzantine quorum-replicated system. In *IEEE International Symposium on Reliable Distributed Systems*, pages 77–89, 2004.

[9] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing protocols for securing group communication. In *Hawaii International Conference on System Sciences*, volume 3, pages 317–326. IEEE Computer Society Press, 1998.

[10] R. Kotla and M. Dahlin. High throughput Byzantine fault tolerance. In *International Conference on Dependable Systems and Networks*, page 575, June–July 2004.

[11] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[12] D. Malkhi, M. K. Reiter, D. Tulone, and E. Ziskind. Persistent objects in the Fleet system. In *DARPA Information Survivability Conference & Exposition II*, pages 126–136, 2001.

[13] P. Narasimhan, K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Providing support for survivable CORBA applications with the Immune system. In *International Conference on Distributed Computing Systems*, pages 507–516, 1999.

[14] M. K. Reiter. The Rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems*, volume 938, pages 99–110. Springer-Verlag, Berlin Germany, 1995.

[15] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *Symposium on Operating Systems Principles*, 2001.

[16] D. Sames, B. Matt, B. Niebuhr, G. Tally, B. Whitmore, and D. Bakken. Developing a heterogeneous intrusion tolerant CORBA system. In *International Conference on Dependable Systems and Networks*, pages 239–248, 2002.

[17] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[18] SOAP specifications. `http://www.w3.org/TR/soap`, April 2005.

[19] S. Tai, T. Mikalsen, and I. Rouvellou. Using message-oriented middleware for reliable web services messaging. In *WES 2003, Springer LNCS 3095*, pages 89–104, 2003.

[20] P. Townend and J. Xu. Replication-based fault tolerance in a grid environment. In *U.K. e-Science 3rd All-Hands Meeting*, August–September 2004.

[21] TPC-W. `http://www.tpc.org/tpcw`, April 2005.

[22] WebServices - Axis. `http://ws.apache.org/axis/`, April 2005.

[23] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson. *Web Services Platform Architecture*. Prentice-Hall, 2005.

[24] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002.

[25] Web services description language (WSDL) 1.1. `http://www.w3.org/TR/wsdl`, April 2005.

[26] Z. Yang. Using a Byzantine-fault-tolerant algorithm to provide a secure DNS. Master's thesis, MIT, Jan. 1999.

[27] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Symposium on Operating Systems Principles*, pages 253–267, October 2003.