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

Sajeeva L. Pallemulle Haraldur D. Thorvaldsson Kenneth J. Goldman

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

Abstract

Mission-critical services must be replicated to guarantee correctness and high availability in spite of arbitrary (Byzantine) faults. Traditional Byzantine fault tolerance protocols suffer from several major limitations. Some protocols do not support interoperability between replicated services. Other protocols provide poor fault isolation between services leading to cascading failures across organizational and application boundaries. Moreover, traditional protocols are unsuitable for applications with tiered architectures, long-running threads of computation, or asynchronous interaction between services.

We present Perpetual, a protocol that supports Byzantine fault-tolerant execution of replicated services while enforcing strict fault isolation. Perpetual enables interaction between replicated services that may invoke and process remote requests asynchronously in long-running threads of computation. We present a modular implementation, an Axis2 Web Services extension, and experimental results that demonstrate only a moderate overhead due to replication.

1 Introduction

Complex distributed applications combine the functionality of services from different providers to perform high-level tasks. Mission-critical services shared by multiple applications must guarantee correct execution and availability in spite of failures. Fail-stop failures, such as host crashes, can be masked using redundant hosts. Achieving *Byzantine Fault Tolerance* (BFT) [1] requires a higher degree of replication[1] since failures may be caused by malicious attacks and arbitrary software or hardware errors. Recent research has yielded practical algorithms [2–5] for Byzantine fault-tolerant execution of passive[2] services.

Replicated shared services must ensure fault isolation between applications by preserving *safety* (consistent state among non-faulty replicas) and *liveness* (eventual execution of correct requests) even when interacting with compromised[3] services. Consequently, mission-critical services require an execution environment that (1) enables interaction between replicated services with different degrees of replication while (2) guaranteeing both safety and liveness.

Prior BFT protocols [2–11] have failed to gain traction due to several major limitations. Some protocols [2–5,9,10] only allow replicated *target* services to be accessed by unreplicated *callers*. A few [6–8, 11] support interaction between replicated services. However, no prior protocol guarantees both safety and liveness of replicated calling services if target services are compromised. Moreover, prior protocols support BFT replication only for passive services. Service oriented architectures (SOA) driven by orchestration (See Section 2.2), however, require support for services with long-running active threads. Application developers increasingly use asynchronous invocation and processing to enable calling services to issue requests in parallel and target services to start processing new requests before previous requests have been fully processed. No prior protocol supports asynchronous invocations from replicated callers and only a few support asynchronous processing at replicated target services. Moreover, prior protocols enforce determinism in applications by precluding access to host specific functions such as local clock queries. Section 3 describes further limitations of existing BFT protocols.

We address these concerns with Perpetual, a practical algorithm for Byzantine fault-tolerant replication of deterministic services. Perpetual supports interaction between services with different degrees of replication. We enforce strict fault isolation between services ensuring both safety and liveness in spite of Byzantine faults. Perpetual supports long-running active threads of computation as well as asynchronous invocation and processing resulting in improved performance and flexibility over prior protocols.

We build upon Perpetual to present Perpetual-WS, middleware that augments the Apache Axis2 [12] Web Service execution environment with a Byzantine fault-tolerant transport module and an API suitable for asynchronous in-

[1] $3f+1$ state machine replicas are needed to tolerate f Byzantine faults.

[2] Passive services modify their state only in response to external calls.

[3] A compromised service has more than f faulty replicas.

vocation and processing. Our benchmark evaluations show that Perpetual-WS scales well to large replica groups and incurs only a modest overhead when used to replicate Web Services that perform non-trivial computation tasks.

The rest of this paper is organized as follows. Section 2 presents relevant background. Section 3 describes unique properties of Perpetual in the context of related work. The Perpetual and Perpetual-WS programming models are presented in Section 5. The architecture and implementation details are discussed in Sections 6 and 7. Section 8 presents macro and micro benchmark evaluations of Perpetual-WS. We conclude in Section 9 with a discussion of future work.

2 Background

Perpetual builds upon the Castro-Liskov Byzantine fault tolerance (CLBFT) [2] algorithm for passive services. We present background on CLBFT, Web Services, and Axis2.

2.1 Castro-Liskov BFT

CLBFT supports passive deterministic BFT services that interact only with unreplicated callers. CLBFT services require $3f + 1$ replicas to tolerate Byzantine faults in up to f replicas. CLBFT also requires messages to be eventually delivered (possibly through retransmissions).

In CLBFT, when a caller sends a request to a designated *primary* replica at a target service, the primary assigns a sequence number to the request and forwards it to other replicas in a *pre-prepare* message. Since the primary may be faulty, the replicas send a *prepare* message to one another to verify they all received the same request and sequence number. Upon receiving $2f$ prepare messages matching the pre-prepare it received from the primary, a replica sends a *commit* message to all replicas. When a replica has matching commit messages from $2f + 1$ replicas, it executes the request and sends the result to the caller. Upon receiving $f + 1$ matching replies, the caller accepts the result.

If a caller times out waiting for a reply (e.g., due to a faulty primary), it sends the request directly to all target replicas. If a replica has not yet received a matching pre-prepare, it forwards the request to the primary and starts a progress timer. If progress under the current primary is unsatisfactory, the replicas switch to a new primary in a *view change* [13] operation. Since view changes are expensive, progress timers adapt to prevent frequent view changes.

Other relevant aspects of CLBFT are summarized in the technical report version of this paper [14].

2.2 Web Services

Web Services use a two-tier model, in which a caller sends a SOAP [15] message to a target Web Service and expects a reply. In practice, the processing of a request may span multiple Web Service tiers across organizational boundaries. For example, when an end-user makes a credit card transaction at an online store, the store Web Service contacts a payment gateway which in turn contacts a bank before authorizing the purchase.

Large enterprise applications are increasingly built by composing Web Services using a Service Oriented Architecture (SOA) [16]. Unlike in tiered applications, where calls to one tier are embedded within another tier, Web Services in SOA applications typically provide unassociated sets of functionality. Applications depend on *orchestrators* that actively execute rules specifying how data flows from one Web Service to another to complete overall tasks. Standards such as the Business Process Execution Language (BPEL) and BPEL engines (e.g., Apache ODE [17]) facilitate the creation and execution of SOA applications.

When a SOAP message is sent to a Web Service synchronously, the caller blocks until it receives a reply message. If the target Web Service is slow to respond, the caller may be blocked needlessly. As a result, asynchronous messaging, in which callers send SOAP messages to target Web Services and continue to execute application logic while waiting for a response, is becoming increasingly popular. New standards (e.g., WS-Addressing [18]) and message exchange patterns (MEP) (e.g., conversational Web Services [19]) have emerged to facilitate asynchronous messaging.

2.3 Apache Axis2

Apache Axis2 [12], a modular open source implementation of SOAP [15], provides API level support for SOAP messaging. Client applications pass messages to the *Axis2 Engine* through the *Client API*. The Axis2 Engine contains a customizable *OUT-PIPE* that holds a series of handlers that may augment the message. Once a message has passed though the OUT-PIPE, it is handed to the *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.

When a message is received by a TransportListener, it is sent to a *MessageReceiver* though the customizable *IN-PIPE* in the Axis2 Engine. At a server, the MessageReceiver may invoke operations and send results back. At a client, the MessageReciver 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

Building upon our previous work [20], we make the following contributions: (1) a version of the Perpetual algorithm that supports asynchronous invocation and processing, (2) a prototype implementation of Perpetual, and (3) the Perpetual-WS extension to support Axis2 Web Services. Figure 1 summarizes our work in the context of related work. In particular, we compare Perpetual to the work of Fry and Reiter (FR) [6], Immune [7], BFT-DNS [8], SWS [11], Thema [9], BFT-WS [10], BASE (CLBFT), [21] and the work of Yin et. al. [22] .

	Perpetual	FR	Immune	BFT-DNS	SWS	Thema	BFT-WS	BASE	Yin
Unreplicated caller – replicated target	*	*	*	*	*	*	*	*	*
Replicated caller - unreplicated target	*	*	*	*	*	*			
Replicated caller – replicated target	*	*	*	*	*				
Fault isolation (Replicated caller safety)	*		*	*					
Fault isolation (Replicated caller liveness)	*								
Long-running threads	*								
Asynchronous invocation	*								
Asynchronous processing	*							*	*
Host-specific information	*							*	
Fast authentication (MAC only)	*					*		*	
Web Services support	*				*	*	*		
Modular implementation	*						*		

Figure 1. Comparison of BFT protocols

Interaction between replicated services: Perpetual, FR, Immune, BFT-DNS, and SWS enable interaction between services with different degrees of replication. Thema and BFT-WS allow replicated services to process requests from unreplicated callers. Thema also allows replicated services to issue calls to unreplicated services. However, Thema and BFT-WS do not support replicated-to-replicated calls.

Fault isolation: Compromised target services may not respond to callers or send different results to different calling replicas. For liveness, the calling replicas may have to deterministically abort the request. For safety, the calling replicas may have to deterministically choose a single result. Perpetual guarantees the safety and liveness of all non-faulty services even when interacting with compromised services. Immune and BFT-DNS guarantee safety but not liveness while FR, Thema, and SWS guarantee neither safety nor liveness when target services are compromised.

Long-running threads of computation: In all protocols, the service being replicated must be deterministic. However, prior protocols require that replicated services be passive as well. Perpetual only requires that the application be single threaded, meaning that active processes such as orchestration can occur within a replicated Web Service application in addition to the processing of external messages.

Asynchronous invocation: Prior protocols only support synchronous invocations by replicated callers, leading to needless blocking at callers. In contrast, Perpetual enables replicated callers to complete requests asynchronously so that they may perform other tasks while waiting for results.

Asynchronous processing: In most prior protocols, the execution of incoming requests is serialized. However, services may have to issue requests to other services and wait for results while processing external requests. Perpetual allows services to process later requests (or perform internal active computations) while waiting for the required results.

Host-specific information: Applications may utilize host specific information (e.g., local clock values, timestamps, and random numbers). Instead of enforcing determinism by precluding access to such information, Perpetual provides methods that return consistent values on all replicas.

Cryptographic overhead: Thema and Perpetual use message authentication codes (MAC) [23] to authenticate messages while BFT-DNS uses digital signatures [24] for some messages. Immune, FR, SWS, and BFT-WS use digital signatures for all messages. MACs can be calculated three orders of magnitude faster than digital signatures.

Modular implementation: In our implementation, low-level cryptographic and transport details are encapsulated within modules that can be easily replaced. BFT-WS also takes advantage of the modularity in Axis2. In contrast, the CLBFT, Thema, and BFT-DNS implementations are tightly coupled with UDP a transport and the SFS [25] cryptographic library. Since Immune only supports BFT CORBA services, it is also tightly integrated within the CORBA framework.

4 Perpetual Algorithm

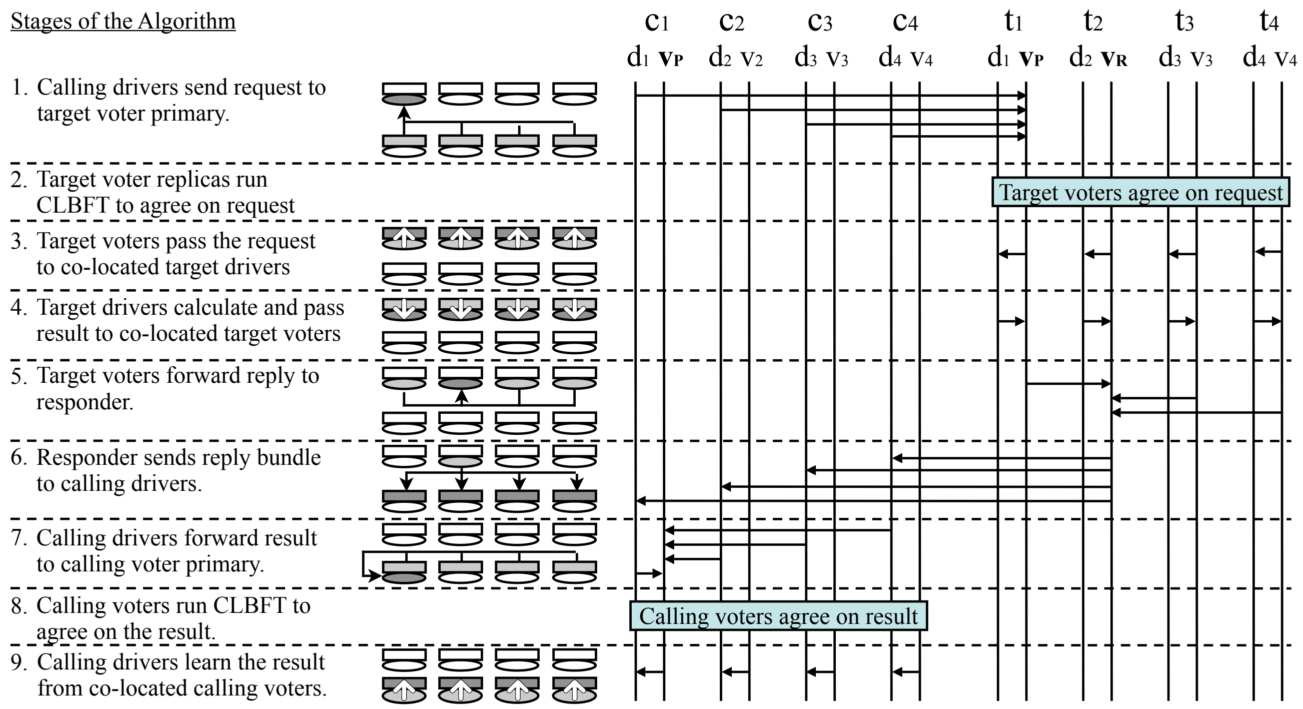
We describe the Perpetual algorithm in terms of a target service t, comprised of $n_t = 3f_t + 1$ replicas $t_1, \ldots, t_{n_t}$, and a calling service c comprised of $n_c = 3f_c + 1$ replicas $c_1, \ldots, c_{n_c}$, where f_t and f_c are the upper bounds on the number of faults tolerated by the target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

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 service s use CLBFT to run agreement on replies to requests originated by s as well as external requests sent to s by other services.

Each driver d_i contains an *executor* e_i, 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 assume that message digests are collision free and that our cryptographic primitives cannot be subverted by an attacker. As in CLBFT, we also assume that messages are eventually delivered to their destinations if retransmitted after exponentially increasing timeouts.

We illustrate the algorithm in Figure 2 by tracing the execution of a request in the non-faulty case. When the executor e_j 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 **(1)**. The voter primary of t waits for at least $f_c + 1$ matching requests before starting CLBFT to agree on the request **(2)**. Upon agreement, each voter v_k of t passes the request to its co-located driver d_k **(3)** using the local event queue. Executor e_k at d_k dequeues the request, executes it, and sends the result back to voter v_k via driver d_k **(4)**. Note that e_k is not required to finish executing a request before starting the execution of the next request. For example, e_k may deterministically choose to start the execution



Stages of the Algorithm
1. Calling drivers send request to target voter primary.
2. Target voter replicas run CLBFT to agree on request
3. Target voters pass the request to co-located target drivers
4. Target drivers calculate and pass result to co-located target voters
5. Target voters forward reply to responder.
6. Responder sends reply bundle to calling drivers.
7. Calling drivers forward result to calling voter primary.
8. Calling voters run CLBFT to agree on the result.
9. Calling drivers learn the result from co-located calling voters.
c1 c2 c3 c4 t1 t2 t3 t4
d1 vP d2 v2 d3 v3 d4 v4 d1 vP d2 vR d3 v3 d4 v4
Target voters agree on request
Calling voters agree on result

Figure 2. The stages of a normal (non-faulty) request. Ellipses show passive voters (v) and rectangles show active drivers (d) of service replicas. Both primaries (P) and the responder (R) of the target voter group are also shown. Source: WUCSE-2007-50 [26]

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* **(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 **(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 **(7)** that uses CLBFT to agree on the reply **(8)**. Once agreement has been reached, each voter of c enqueues the result in the local event queue **(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.

4.1 Fault Handling

Driver d_j of calling replica c_j starts an *operation timer* upon sending a request to t. If the timer expires before a reply is received, there are three possible explanations: (1) The voter primary of t is faulty and discarded the request; (2) The responder of t is faulty and didn't send the reply to some or all of the calling replicas; or (3) The timeout value is too low for current network conditions.

When d_j times out waiting for a reply, it re-sends the request to all n_t voters of t. If a voter v_k of target replica t_k receives at least $f_c + 1$ matching requests, it checks whether its primary has started agreement on the request. If not, v_k forwards the request (including the bundle of $f_c + 1$ signatures) to its primary. It also starts a *progress timer*, as defined in CLBFT. Once e_k has successfully executed the operation (potentially under a new primary at t), voter v_k multicasts the reply to all drivers of c. If no more than f_t target replicas are faulty, each driver of c eventually receives at least $f_t + 1$ matching replies, as in the normal case.

4.1.1 Compromised Calling Group

If c is compromised (more than f_c faulty replicas), it should not be able to violate the correctness of t. Since the voters of t only start agreement upon receiving matching requests from at least $f_c + 1$ different replicas of c, the case of a compromised calling group c reduces to the case of a single faulty unreplicated caller in CLBFT, and safety is ensured.

4.1.2 Compromised Target Group

If t is compromised, it should not be able to violate the correctness of c nor inhibit its progress. Replicas of c individually accept replies from t upon receiving $f_t + 1$ valid signatures for it, but t could send quorums of replies with different result values to different replicas of c. If at least $f_c + 1$ replicas of c receive the same valid reply, then that reply may eventually be voted upon by the voters of c and

```
interface BFTAdapter (Perpetual):              interface MessageHandler (Perpetual-WS):                            // Messaging APIs
  Invocation sendAsyncReq(Message req);          void send(MessageContext req);                                // Asynchronous request
  Invocation receiveReply();                     MessageContext receiveReply();                                 // Fetch next reply
  MessageBuffer sendSyncReq(Message req);        MessageContext sendReceive(MessageContext req);              // Synchronous request
  Message receiveRequest();                      MessageContext receiveRequest();                             // Fetch next request
  void sendReply(MessageBuffer rep);             void sendReply(MessageContext rep);                                // Send reply
                                                 MessageContext receiveReply(MessageContext req);    // Fetch reply for request
```

Figure 3. The Perpetual and Perpetual-WS APIs provide messaging support.

placed in the event queue for consumption by all executors of c. However, if t does not send the same reply to at least f_c+1 replicas of c, the executors of c may deadlock waiting for a reply that will never arrive. To preserve liveness, we allow each driver of c to send an *abort request* to the voter primary of c. If at least f_c+1 calling replicas send abort requests, then the abort request may be voted upon and placed in the event queue instead of a reply from t.

Further details on fault handling, checkpointing, garbage collection, and recovery as well as the formal I/O automata [27] model are included in the technical report version [14].

5 Perpetual Programming Model

Perpetual supports applications implemented using a single ongoing thread of computation. We do not distinguish between callers and targets. Instead, applications deployed using Perpetual may (1) issue requests, (2) query for incoming requests, (3) query for incoming replies, and (4) issue replies. The Perpetual API shown in Figure 3 supports this programming model. Further details of all interfaces (including checkpointing, garbage collection, and state transfer) are included in the technical report version [14].

5.1 Perpetual API

A service may send asynchronous requests using the `SendAsyncReq` method that returns an `Invocation` object. `Invocation`s implement the `java.util.concurrent.Future` interface, parameterized by `MessageBuffer`[4]. The caller may call `get` on the `Invocation` at a later time to get the results of the operation. If the result is not yet available, the call blocks until it arrives. If the operation is aborted, an `OperationAbortedException` will be thrown.

An application may attempt to abort a pending request by calling `cancel` on the corresponding `Invocation`. Calling `get` with a timeout value will either return the operation's result or abort it after waiting for at least that amount of time. It is important to note that calling `cancel` only signals the preference for a request to be aborted. The ultimate fate of a request is determined by agreement within the voter group. Hence, it is necessary to call `get` on every `Invocation` to learn its outcome.

The method `SendSyncReq` implements synchronous calls for convenience. For fully asynchronous calls a caller may request the next available result for *any* pending request using the `receiveReply` method. The method returns the next available reply from the event queue, blocking, if necessary, until some reply is available.

A service that accepts incoming requests may use the `receiveRequest` method to obtain the next external request and the `sendResult` method to send replies.

[4]`MessageBuffer`s wrap immutable `java.nio.ByteBuffer`s.

5.2 Perpetual-WS API

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 way that is transparent to the software is beyond the scope of this paper. Instead, we currently support the MessageHandler API shown in Figure 3.

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.

Since `MessageContext` does not implement `java.util.concurrent.Future`, we include the `receiveReply(MessageContext request)` method to enable blocking for a result of a particular request. To abort a request in Perpetual-WS, the caller must specify a timeout within the `MessageContext` of the request.

5.3 Utility Methods

The Perpetual API provides several utility methods. When `currentTimeMillis` or `timestamp` is called, a request is issued to the voter group to obtain a consistent value. The primary of the voter group suggests a value to be agreed upon by all the voters. Calls to `random` return `java.util.Random` objects with an agreed upon seed.

6 Perpetual Architecture

We describe the Perpetual architecture and the Perpetual-WS extension by tracing the execution of a request issued by an Axis2 Web Service during fault-free execution, as shown in Figure 4. We omit interfaces related to checkpointing, state transfer, and garbage collection in order to reduce complexity. Further details on the content of messages may be found in the technical report version [14].

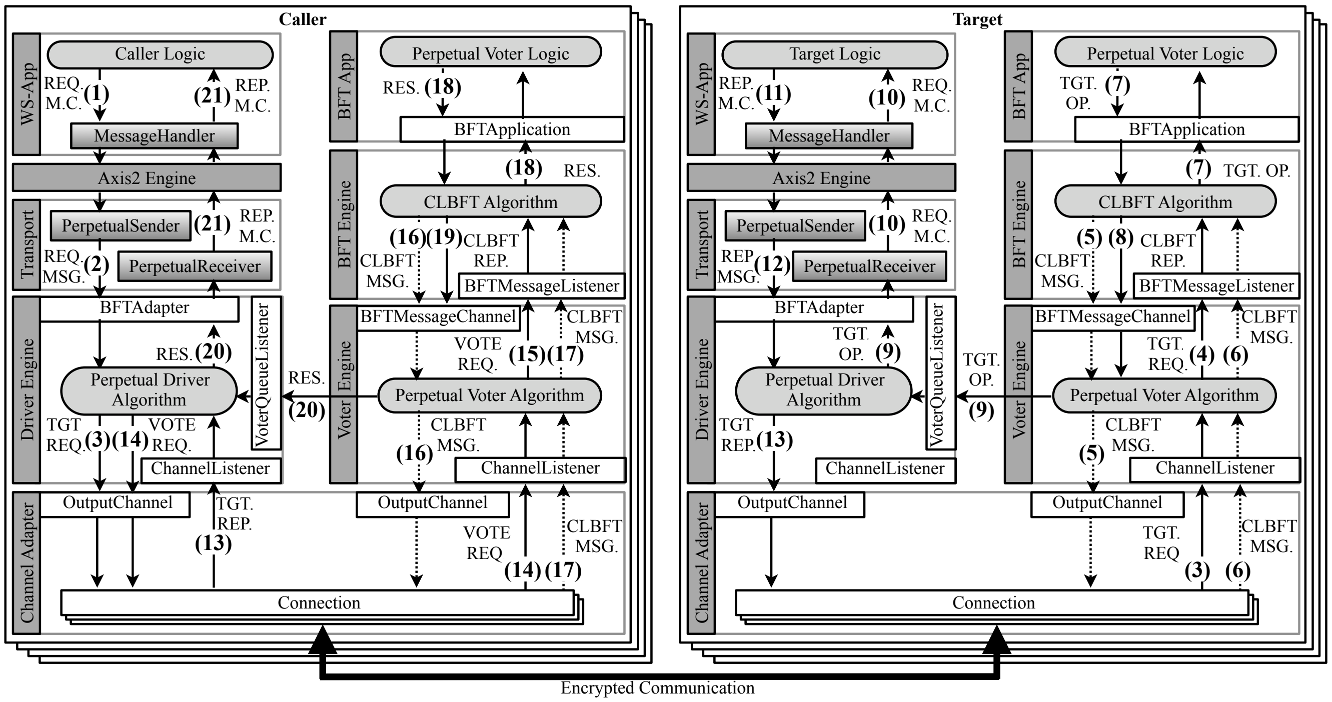


Figure 4. The Perpetual architecture and the Perpetual-WS extension: High-level modules (darkly shaded on the left), interfaces (rectangles at module edges), algorithm specific sub-modules (rectangles with rounded-edges), and Axis2 specific sub modules (shaded rectangles with a gradient) are shown. The numbered arrows indicate the flow of messages during fault-free execution.

6.1 Modular Interaction

The calling logic issues a request to a target Web Service by passing a `MessageContext` to the MessageHandler to initiate the request **(1)**. The `MessageContext` is then sent through the Axis2 Engine to the PerpetualSender that implements the Axis2 `TransportSender` interface. To support non-blocking calls, the sending thread must not block waiting for a reply from the Perpetual layer. Hence, the `Message` object (created using the `MessageContext`) is passed to the driver using the non-blocking `sendAsyncReq` method of the BFTAdapter API (2).

The Perpetual Driver Algorithm uses the `Message` object to create a Target Request message, which is passed to the ChannelAdapter module using the OutputChannel interface **(3)**. The ChannelAdapter adds authentication data and sends the resulting message to the voter of the target primary using an encrypted channel.

The ChannelAdapter at the target primary receives and authenticates the message before passing the enclosed Target Request up to the VoterEngine module, using the ChannelListener interface. The Perpetual Voter Algorithm encapsulated within the VoterEngine collects at least $f_c + 1$ (where c is the caller) matching Target Requests before sending the Target Request to the BFTEngine module using the BFTMessageListener interface **(4)**. Once CLBFT agreement has been reached on the Target Requet **(5, 6)**, the BFTEngine passes the Operation to the Application module through the BFTApplication interface **(7)**. The Perpetual voter logic returns the Target Operation back to the CLBFTEngine, which wraps the Target Operation in a CLBFT Reply and passes it to the VoterEngine through the VoterMessageChannel interface **(8)**. The CLBFT Reply is intercepted and the Target Operation contained within it is extracted and forwarded to the DriverEngine using the VoterQueueListener interface **(9)**.

The PerpetualReceiver fetches the `Message` object for the Target Operation, extracts the `MessageContext` and passes it to the MessageHandler through the Axis2 Engine **(10)**. 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 `receiveRequest` method of the MessageHandler API.

When it is ready to send a reply, the target logic calls the `sendReply` method of the MessageHandler, passing in the `MessageContext` of the reply **(11)**. The `MessageContext` is then sent to the Transport layer through the Axis2 Engine. The PerpetualSender constructs a `Message` object and passes it to the DriverEngine through the `sendReply` method of the BFTAdapter API **(12)**.

The DriverEngine uses the `Message` object to construct

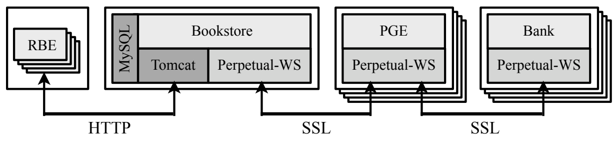


Figure 5. The TPC-W Configuration

a Target Reply message and sends it to the ChannelAdapter through the OutputChannel interface **(13)** to be sent (possibly via a designated responder) to the calling drivers.

Each DriverEngine at the caller collects at least $f_t + 1$ (where t is the target service) matching Target Replies before sending a Vote Request containing the Result to the primary of the calling voter group **(14)**. Stages **(14)** through **(20)** mirror stages **(3)** through **(9)** exactly.

The PerpetualReceiver fetches the `Message` for the Result, extracts the `MessageContext`, and passes it to the MessageHandler, through the Axis2 Engine. If the original request was synchronous, the MessageHandler returns the reply `MessageContext` to the blocked caller thread **(12)**. Otherwise, the `MessageContext` is placed in a FIFO queue to be fetched by the caller thread.

7 Implementation

Implemented in Java, the Perpetual code base is organized into three libraries containing the code for CLBFT, Perpetual, and Communication. All the code (including the Perpetual-WS extension library) is available online [28].

The CLBFT Library: The BASE implementation of CLBFT is not modular, coupling the CLBFT algorithm with low-level (connections, cryptography) concerns[5]. Therefore, we implemented CLBFT in Java abstracting away cryptography and network functions through interfaces.

The Perpetual Core Library: The Perpetual Core library contains the implementations of the DriverEngine and VoterEngine modules. As with the CLBFT library, cryptography and network functions are delegated elsewhere.

The Communications Library: The ChannelAdapter (CA) module contained in the Communications library guarantees exactly-once (possibly out-of-order) delivery of messages between correct replicas even when connections break and are re-established. Replicas are identified with RSA public keys, which are used to establish SSL sessions. The CA includes a MAC authenticator [23] for each end-destination recipient replica of a message to support forwarding without digital signatures.

The Perpetual-WS Library: The Perpetual-WS library contains the implementations of the PerpetualSender, PerpetualReceiver, and the MessageHandler. In addition, the library contains a simple name service resolution mechanism as described in the technical report version of this paper [14].

[5]BASE also needs the discontinued SFS [25] library

8 Experiments

We conducted both macro and micro benchmark evaluations of Perpetual-WS to ascertain the scalability and efficiency of our implementation. As our macro-benchmark we used an open source implementation [29–31] of the TPC-W [32] web e-Commerce benchmark. For our micro-benchmarks, we used a two-tier setting and measured the throughput of the calling service.

8.1 TPC-W Benchmark

As shown in Figure 5, the TPC-W benchmark models a multi-tiered e-commerce application. The benchmark measures the throughput of an online bookstore with 14 web pages in Web Interactions Per Second (WIPS). Remote Browser Emulators (RBE) are used to simulate end-users. The RBEs use different workloads to simulate user behavior biased toward browsing, shopping, or ordering. When using the ordering workload, approximately 10% of total traffic received by the bookstore results in requests to an external Payment Gateway Emulator (PGE).

Our setup mirrors the setup used to evaluate Thema [9]. All the RBEs executed within a single host with default settings. The RBEs used the ordering workload to issue requests to the bookstore Web Service over HTTP connections. The bookstore Web Service was deployed on another host and used an Apache Tomcat Servlet engine and a (co-located) MySQL database. Since the TCP-W implementation did not include a PGE, we changed the bookstore to call a PGE Web Service implemented using Perpetual-WS. The PGE calls another Perpetual-WS Web Service that simulates the actions of a credit card issuing bank. We utilized four different configurations where the PGE and Bank Web Services both executed in replica groups of varying size. We disregarded the minimum execution time requirement for the PGE to ensure that the effects of replication were not masked. Both the PGE and Bank Web Services used asynchronous messaging.

8.2 Micro-benchmarks

Our micro-benchmarks used a two-tier setting with caller and target Web Services both implemented using Perpetual-WS. Measurements were recorded at the calling Web Service. We first measured the request throughput as the number of calling and target Web Service replicas was varied. We then performed experiments to evaluate the effects of non-zero processing time and the performance gains made by using asynchronous requests. To simulate null-operations, we implemented a simple increment method at the target Web Service. To simulate non-zero execution time, we used digest calculations calibrated to take the required length of time. We measured time taken to complete 500 calls to calculate each data point. The first 20 measurements were discarded to account for startup costs.

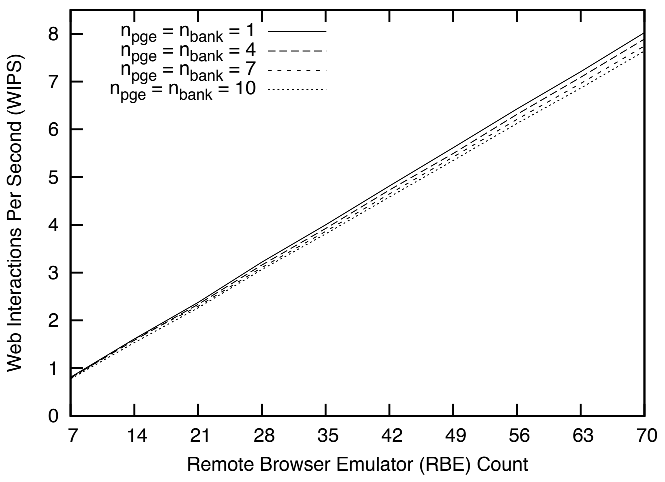


Figure 6. TPC-W benchmark results

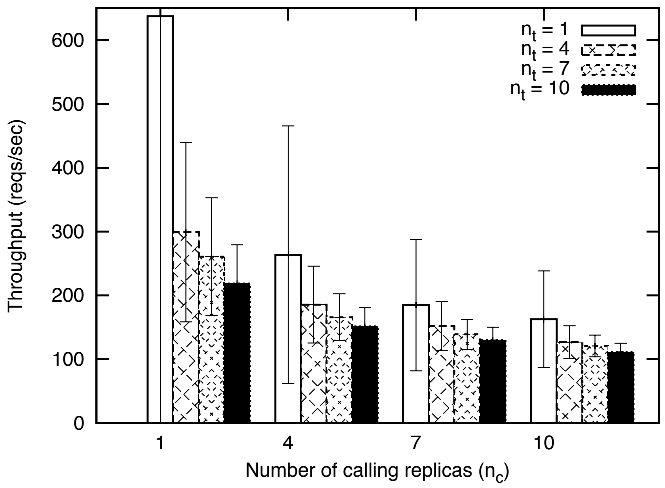


Figure 7. Replica scalability (Null requests)

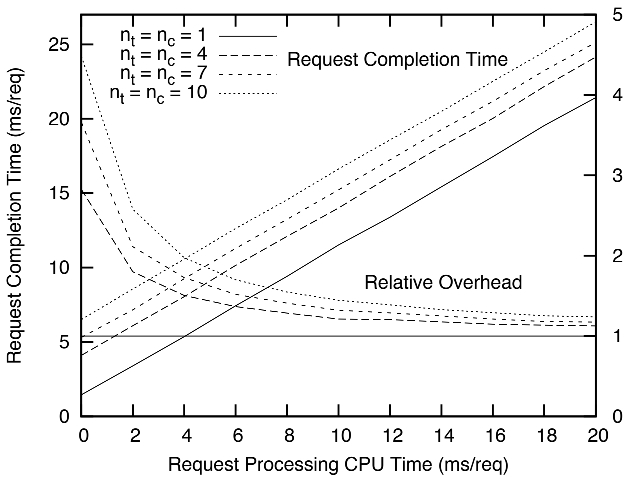


Figure 8. Effect of non-zero processing time

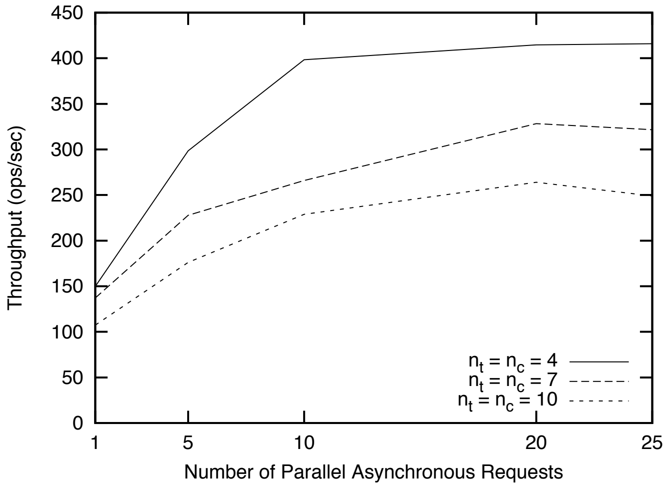


Figure 9. Effect of asynchronous messaging

8.3 Experimental Setup

All of our experiments were performed on a dedicated Washington University testbed [33] made up of 2GHz Opteron machines with 512 MB of RAM, connected via a Netgear GSM7352S Gigabit Ethernet router (with the `ping` tool reporting $78\mu s$ 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.

8.4 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 about 10% of all requests to the Bookstore resulted in calls to the PGE these gains represent a significant improvement in performance.

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 argues well for the scalability of Perpetual-WS.

Figure 8 shows the effect on throughput when incoming requests take non-zero time to process, shown relative to the case with no replication. We can see that as requests take longer to process, the overhead of replication rapidly decreases. For example, in the case of four replicas in both the caller and target replica groups, the throughput increases from 30% (of the no replication case) for null operations to 73% 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 270%, 239%, and 246%, respectively, when asynchronous messaging was used.

9 Conclusion

We plan to generalize the BFT Engine module to encapsulate BFT state machine replication algorithms [5] other than CLBFT. We also plan to extend the capabilities of Perpetual-WS to include execution of BPEL [34] processes using the Apache ODE [17] execution engine.

We thank Charlie Wiseman and the rest of the ONL team. The authors were supported in part by National Science Foundation grants 0305954, 0618266, and 0722328.

References

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

[2] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proc. 3rd Symp. on Operating Systems Design and Implementation*, pages 173–186, 1999.

[3] J. Cowling et. al. HQ replication: a hybrid quorum protocol for byzantine fault tolerance. In *Proc. of the 7th Symp. on Operating systems design and implementation*, pages 177–190, 2006.

[4] M. Abd-El-Malek et. al. Fault-scalable Byzantine fault-tolerant services. *SIGOPS Operating Systems Review*, 39(5):59–74, 2005.

[5] R. Kotla et. al. Zyzzyva: speculative byzantine fault tolerance. In *Proc. of 21st ACM SIGOPS Symp. on Operating Systems Principles*, pages 45–58, 2007.

[6] C. Fry and M. Reiter. Nested Objects in a Byzantine Quorum-Replicated System. In *Proc. 23rd Intl. Symp. on Reliable Distributed Systems*, pages 79–89, 2004.

[7] P. Narasimhan et. al. Providing Support for Survivable CORBA Applications with the Immune System. In *Proc. 19th Intl. Conf. on Distributed Computing Systems*, pages 507–516, 1999.

[8] S. Ahmed. A Scalable Byzantine Fault Tolerant Secure Domain Name System, 2001. Master's thesis, MIT.

[9] M. G. Merideth et. al. Thema: Byzantine-Fault-Tolerant Middleware forWeb-Service Applications. In *Proc. 24th Symp. on Reliable Distributed Systems*, pages 131–140, 2005.

[10] W. Zhao. BFT-WS: A Byzantine Fault Tolerance Framework for Web Services. In *Proc. Middleware for Web Services Workshop*, 2007.

[11] W. Li et. al. A Framework to Support Survivable Web Services. In *Proc. of the 19th IEEE Intl. Parallel and Distributed Processing Symp.*, pages 93–102, 2005.

[12] S. Perera et. al. Axis2, Middleware for Next Generation Web Services. In *Proc. of the IEEE Intl. Conf. on Web Services*, pages 833–840, 2006.

[13] M. Castro. Practical Byzantine Fault Tolerance. Technical Report MIT-LCS-TR-817, MIT, 2001.

[14] S. L. Pallemulle, H. D. Thorvaldsson, and K. J. Goldman. Byzantine Fault-Tolerant Web Services for n-Tier and Service Oriented Architectures. Technical Report WUCSE-2007-53, Washington University, 2007.

[15] W3C. *SOAP Version Messaging Framework*, 1.2 edition, June 2003.

[16] E. Newcomer and G. Lomow. *Understanding SOA with Web Services (Independent Technology Guides)*. Addison-Wesley Professional, 2004.

[17] Apache Software Foundation. *Apache Orchestration Director Engine (ODE) Architectural Overview*, October 2007.

[18] W3C. *Web Services Addressing (WS-Addressing)*, 1.1 edition, August 2004.

[19] W3C. *Web Services Conversation Language (WSCL)*, 1.0 edition, March 2002.

[20] S. Pallemulle, I. Wehrman, and K. Goldman. Byzantine Fault Tolerant Execution of Long-running Distributed Applications. In *18th IASTED Paralell and Distributed Computing and Systems*, pages 528–534.

[21] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using Abstraction to Improve Fault Tolerance. In *Proc. 18th Symp. on Operating Systems Principles*, pages 15–28, 2001.

[22] J. Yin et. al. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *Proc. of the 19th ACM Symp. on Operating Systems Principles*, pages 253–267, 2003.

[23] B. Prenel and P. van Oorschot. MDx-MAC and Building Fast MACs from Hash Functions. In *Proc. 15th Conf. on Advances in Cryptology*, pages 1–14, 1995.

[24] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[25] D. Maziores et. al. Separating Key Management from File System Security. In *Proc. 17th Symp. on Operating systems principles*, pages 124–139, 1999.

[26] S. L. Pallemulle, H. D. Thorvaldsson, and K. J. Goldman. Perpetual: Byzantine Fault Tolerance for Multi-Tiered Distributed Applications. Technical Report WUCSE-2007-50, Washington University, 2007.

[27] N. A. Lynch and M. R. Tuttle. Hierarchical Correctness Proofs for Distributed Algorithms. In *Proc. 6th ACM Symp. on Principles of Distributed Computing*, pages 137–151, 1987.

[28] Distributed Systems Group, Washington University in St. Louis. *The SWFTI Project*, March 2008.

[29] Todd Bezenek et. al. Characterizing a Java Implementation of TPC-W. In *Proc. of the 3rd Workshop On Computer Architecture Evaluation Using Commercial Workloads*, January 2000.

[30] J. Kiefer. *TPC-W Java Implementation*, May 2005.

[31] ObjectWeb Consortium. *TPC-W Benchmark*, 1.0 edition, February 2005.

[32] Daniel A. Menascé. TPC-W: A Benchmark for E-Commerce. *IEEE Internet Computing*, 6(3):83–87, 2002.

[33] J. DeHart et. al. The Open Network Laboratory. *SIGCSE Proceedings*, Mar 2006.

[34] IBM Inc. *Business Process Execution Language for Web Services*, 1.1 edition.