Department of Computer Science & Engineering

2007-50

Perpetual: Byzantine Fault Tolerance for Federated Distributed Applications

Authors: Sajeeva L. Pallemulle, Haraldur D. Thorvaldsson, and Kenneth J. Goldman

December 10, 2007

Corresponding Author: kjg@cse.wustl.edu

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160

2007-50

Abstract

Modern distributed applications rely upon the functionality of services from multiple providers. Mission-critical services, possibly shared by multiple applications, must be replicated to guarantee correct execution and availability in spite of arbitrary (Byzantine) faults. Furthermore, shared services must enforce strict fault isolation policies to prevent cascading failures across organizational and application boundaries. Most existing protocols for Byzantine fault-tolerant execution do not support interoperability between replicated services while others provide poor fault isolation. Moreover, existing protocols place impractical limitations on application development by disallowing long-running threads of computation, asynchronous operation invocation, and asynchronous request processing.

We present Perpetual, a protocol that facilitates unrestricted interoperability between replicated services while enforcing strict fault isolation criteria. Perpetual supports both asynchronous operation invocation and asynchronous request processing. Perpetual also supports long-running threads of computation, enabling Byzantine fault-tolerant execution of services that carry out active computations. We present performance evaluations demonstrating a moderate overhead due to replication.

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160

Perpetual: Byzantine Fault Tolerance for Federated Distributed Applications

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

Modern distributed applications rely upon the functionality of services from multiple providers. Mission-critical services, possibly shared by multiple applications, must be replicated to guarantee correct execution and availability in spite of arbitrary (Byzantine) faults. Furthermore, shared services must enforce strict fault isolation policies to prevent cascading failures across organizational and application boundaries. Most existing protocols for Byzantine fault-tolerant execution do not support interoperability between replicated services while others provide poor fault isolation. Moreover, existing protocols place impractical limitations on application development by disallowing long-running threads of computation, asynchronous operation invocation, and asynchronous request processing.

We present *Perpetual*, a protocol that facilitates unrestricted interoperability between replicated services while enforcing strict fault isolation criteria. Perpetual supports both asynchronous operation invocation and asynchronous request processing. Perpetual also supports long-running threads of computation, enabling Byzantine fault-tolerant execution of services that carry out active computations. We present performance evaluations demonstrating a moderate overhead due to replication.

KEY WORDS
Byzantine fault tolerance, replicated services, long-running threads, asynchronous invocation, asynchronous processing

1 Introduction

Fault tolerance requires replication. Mission-critical services must handle failures at time scales too short for human oversight. Fail-stop failures, such as host crashes, can be handled by switching to auxiliary host(s). However, a host under the control of an attacker may work to disrupt the application, possibly in collusion with other hosts. Although tolerating such arbitrary (Byzantine) faults is more expensive[1] than tolerating crash failures, recent research has yielded practical algorithms [1, 2] for Byzantine fault-tolerant execution of deterministic state machines. Most notably, Castro and Liskov [1] have presented a protocol for Byzantine fault-tolerant replication of passive services that update their state in response to external requests.

Growing adoption of service oriented architectures (SOA) [3] has resulted in services that perform common tasks for multiple distributed applications. Such services range from simple mapping services (e.g., Google Maps) [4] to mission-critical services such as payment gateways (e.g., Google Checkout [5], Amazon FPS [6]). Previous attempts to support Byzantine fault-tolerant execution of mission-critical services have failed to gain traction due to several major limitations:

Service interoperability: Most prior protocols, including CLBFT, do not support replicated *calling* services that issue requests to *target* services that process requests. Similarly, most prior protocols do not support target services that process requests from replicated calling services. SOA applications require interoperability between services regardless of their degrees of replication.

Safety and liveness: Even Byzantine fault-tolerant replicated services may become *compromised*[2] over time. Most prior protocols guarantee *safety*[3] and *liveness*[4] of target services even when calling services are compromised. However, no prior protocol guarantees both safety and liveness of calling services if target services are compromised. Consequently, a compromised target service can violate safety of a calling service by sending different results to different calling replicas. Moreover, it can violate liveness of a calling service by not responding to outstanding requests. This model is clearly inadequate for services shared among many distributed applications since faults can propagate across organizational and application boundaries. If all calling replicas can be guaranteed to agree upon the same result values (safety), the calling logic can use application level verification strategies (e.g., queries to multiple target services) to establish the integrity of result values (and take compen-

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

[2] A compromised Web Service has more than f faulty replicas.

[3] If state of all non-faulty replicas is identical after each execution step.

[4] If scheduled tasks execute eventually.

satory actions if necessary). If requests issued to unresponsive target services can be deterministically aborted by all non-faulty calling replicas (liveness), the calling logic can issue requests to alternate target services instead.

Programming model: BFT protocols only support deterministic services. However, prior BFT protocols place an additional restriction by disallowing long-running computations at replicated services. This model is inadequate to support services that perform a variety of active mission-critical tasks (e.g., network monitoring, Web Services orchestration) in long-running threads of computation.

Performance: BFT replication requires multiple rounds of voting, increasing operation latency. Asynchronous invocation by replicated callers allows services to continue to process pending tasks instead of being blocked waiting for results. To process incoming requests, a target service may need to issue requests to another service and wait for results. If asynchronous processing is supported, then target services may deterministically choose to complete such requests at a later time and start processing other requests. By supporting both asynchronous invocation and asynchronous processing, the throughput of replicated services can be increased substantially. No prior approach supports asynchronous invocation or processing.

This paper presents Perpetual, a practical solution for Byzantine fault-tolerant replication of deterministic services. Perpetual enables interaction between replicated services with any degree 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, leading to improved coverage and performance over prior protocols. This paper contributes (1) significant modifications of our previous work [7] to support asynchronous invocation and processing to mask latency, (2) an efficient extensible implementation using Java, and (3) thorough performance evaluation experiments on both a local area network and the PlanetLab [8] research network.

An immediate use for Perpetual would be in the management of networks and distributed applications. These types of systems are growing more complex and costly to administer and are increasingly targeted by attackers [9, 10]. These factors have spurred research into systems and networks regulated by highly autonomous configuration management and monitoring services [11, 12, 13]. Fault-tolerance concerns argue against centralizing such functions, since the failure of a management service may put an entire enterprise at risk. However, it is fundamentally difficult to design decentralized algorithms for tasks such as resource allocation, load balancing, and intrusion detection. Hance, logically centralized control algorithms actively executing on replicated Byzantine fault tolerant services are an attractive alternative.

The rest of this paper is organized as follows. Section 2 compares Perpetual with prior work in the field. Section 3 provides background information before we present the Perpetual algorithm in Section 4. Sections 5, 6, and 7 describe the architecture, API, and main design choices in our implementation. Section 8 shows the results of our microbenchmarks. We conclude in Sections 9 and 10 with a summary and directions for future work.

2 Related Work

We build upon the Castro and Liskov Byzantine Fault Tolerance (CLBFT) algorithm [1] to develop a practical, efficient, and comprehensive protocol for BFT execution of services. In this section, we present unique properties of Perpetual in the context of related work. In particular, we compare Perpetual to the work of Fry and Reiter [14], Immune [15], SWS [16], BFT-DNS [17], and Thema [18].

Replicated service interoperability: Perpetual, Immune, and SWS enable unrestricted interoperability among services with differing degrees of replication. In contrast, CLBFT only supports replicated services that process requests from unreplicated callers. Unlike Immune, Perpetual requires only a constant number of message hops per request regardless of replica group sizes, leading to better performance in large replica groups. Thema enables replicated callers to issue requests to an unreplicated targets, but does not support interaction between replicated services. The approach of Fry and Reiter is a quorum-replication technique that only supports simple read/write operations, whereas Perpetual supports state machine replication.

Fault isolation: Perpetual guarantees the safety and liveness of all non-faulty services even when interacting with potentially compromised services. Immune, Thema, and BFT-DNS guarantee both safety and liveness of target services when calling services are compromised whereas SWS does not guarantee safety or liveness of target services if calling services are compromised. BFT-DNS and Immune only guarantee safety of calling services when target services are compromised. Both Thema and SWS guarantee neither safety nor liveness of calling services when target services are compromised.

Asynchronous invocation: Similar to prior protocols, Perpetual supports synchronous requests where a calling service must wait for the result of the previous request before issuing another request to the same target service. In addition, Perpetual also supports asynchronous requests where calling services may achieve higher throughput by issuing new requests without waiting for results of outstanding requests. No other approach (including CLBFT) supports asynchronous requests.

Asynchronous processing: In Immune, SWS, BFT-DNS, and Thema, the processing of incoming requests is serialized. However, services may have to issue requests to other services and wait for results while processing external requests. Perpetual enables the service to process other requests and replies (or perform internal active computations) while waiting for results.

Efficient message authentication: Perpetual employs message authentication codes (MAC) [19] to authenticate all regular communication between replicas. Consequently, Perpetual enjoys a significantly lower authentication delay than Immune, SWS, and BFT-DNS that employ full digital signature schemes in some or all of their communication. The approach of Fry and Reiter also has low scalability due to exclusive use of digital signatures and an exponential growth in the cost of authentication for each nested step in remote invocations.

Modular implementation: As shown in Section 5, the Perpetual implementation is highly modular. Our implementation is not tightly coupled with any cryptographic or transport technology. Such low-level concerns are encapsulated within modules that can be replaced easily. In contrast, the CLBFT, Thema, and BFT-DNS implementations are tightly coupled with a UDP transport medium and the SFS [20] cryptographic library. Since Immune only seeks to provide Byzantine fault tolerance to CORBA [21] servers it is also tightly integrated within the CORBA framework. Implementation details are not avialable for SWS.

3 Background

The Castro-Liskov Byzantine fault tolerance (CLBFT) [1] algorithm supports passive deterministic target services that interact only with unreplicated callers. CLBFT services require $3f + 1$ replicas to tolerate Byzantine faults in up to f replicas. Messages between replicas can be delayed, reordered, duplicated, or retransmitted but must be eventually delivered.

In CLBFT, when a caller sends a request to a designated *primary* target replica, 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 each other to ensure 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 (including itself), 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 original request directly to all target replicas. If a target replica has not yet received a corresponding pre-prepare message, it forwards the request to its primary and starts a progress timer. If progress under the current primary is unsatisfactory, the target replicas switch to a new primary in a *view change* operation. Since view changes are expensive, progress timers adapt to prevent frequent view changes.

4 Perpetual Algorithm

Perpetual enables a replicated service to both accept requests from and issue requests to any number of other replicated services (possibly of size one). For ease of exposition, though, we describe the algorithm in terms of a target service t, comprised of $n_t = 3f_t + 1$ replicas $t_1, \ldots, t_{n_t}$, and a calling service c comprised of $n_c = 3f_c + 1$ replicas $c_1, \ldots, c_{n_c}$, where f_t and f_c are the upper bounds on the number of faulty replicas tolerated by the target and calling services, respectively. Our formal I/O Automata [22] model, assumptions, and proof sketch may be found in [23].

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 agree on replies to requests originated by s as well as external requests sent to s by other services.

Each driver contains an *executor*, a black box capturing application behavior. Executors model deterministic applications that: (1) request operations on target services and process their replies and (2) execute operations requested by calling services and send back replies. The requests may be synchronous (blocking) or asynchronous (non-blocking). Since executors are deterministic, we are guaranteed that if two correct executors (of the same replica group) have gone through the same sequence of requests and replies, the next request or reply issued by those executors will match.

Replicas of a particular replica group may receive external requests, or replies for outstanding requests at different times. Yet we require their executors to consume all incoming requests and replies in the same exact order. Additionally, since some replicas in the group may be faulty or external services may be compromised, replicas must carry out Byzantine agreement on both reply values and external requests. Both of these requirements are met by the voters. Each voter participates in the agreement on replies and external requests and places them in a local *event queue* in the order of agreement. Each queue is consumed by the local executor in a blocking operation. Since all executors issue the same sequence of requests (including requests to dequeue items from the event queue), all executors of non-faulty replicas process the same external requests or outstanding replies at the same points in their executions.

4.1 Normal Operation

We illustrate the algorithm in Figure 1 by tracing the execution of a request in the non-faulty case. When the executor at calling replica c_j requests an operation to be performed by t, the driver d_j will send the request to the voter

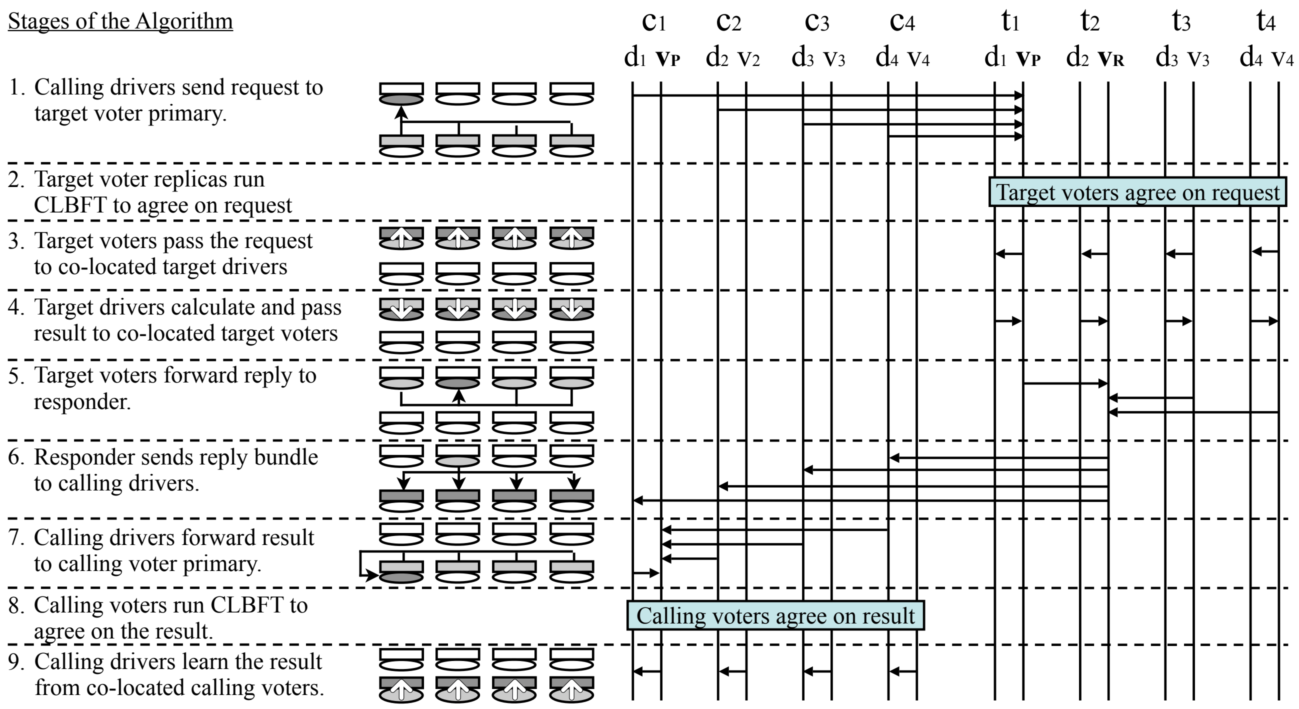
Stages of the Algorithm
1. Calling drivers send request to target voter primary.
2. Target voter replicas run CLBFT to agree on request
3. Target voters pass the request to co-located target drivers
4. Target drivers calculate and pass result to co-located target voters
5. Target voters forward reply 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 1. The stages of a normal (non-faulty) request. Ellipses show passive voters (v) and rectangles show active drivers (d) of service replicas. The primaries (P) of both voter groups and the responder (R) of the target voter group are also shown.

primary of t (Stage 1). The voter primary of t will wait for at least $f_c + 1$ matching requests before starting CLBFT to agree on the request (Stage 2). Upon agreement, each voter v_k of t will pass the request to its co-located driver d_k (Stage 3) using the local event queue. The executor at d_k will dequeue the request, execute it (possibly by issuing external requests) and send the result back to voter v_k via driver d_k (Stage 4). Note that the executor at d_k is not required to finish executing a request before starting the execution of the next request. The executor at d_k, for example, may deterministically choose to start the execution of the next request while waiting for replies to external requests that were issued during the execution of the previous request.

Under CLBFT, each replica of t would send its reply directly to c. However, c is replicated and we wish to avoid the $n_t * n_c$ messages that would result from having all voters of t send replies to all drivers of c. Therefore, each voter of t will forward its reply to a particular voter of t, known as the *responder* (Stage 5). The responder, specified in the original request messages from the drivers of c, will collect $f_t + 1$ matching replies and forward the reply bundle to each driver of c (Stage 6). When a driver d_j of calling replica c_j receives this message, it will authenticate the reply bundle and forward the result to the primary of c's voter group (Stage 7) that will use CLBFT to agree on the reply (Stage 8). Once agreement has been reached, each voter of c will enqueue the result in the local event queue (Stage 9). When an executor of c deterministically decides to consume the result of a request, it will pull that result from the event queue, blocking if necessary until a result for that request is available in the queue.

4.2 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 matching requests from at least $f_c + 1$ different calling drivers, 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 *view-change timer*, as defined in CLBFT. Once the executor at target replica t_k has successfully executed the operation (potentially under a new primary among the voters of t), voter v_k multicasts the reply to all drivers of c. If there are no more than f_t faulty target replicas, each driver of c eventually receives at least $f_t + 1$ matching replies, and the reply is then processed as in the normal case.

4.2.1 Compromised Calling Group

If c is compromised (if more than f_c replicas of c are faulty), it should not be able to violate the correctness of t. Since the

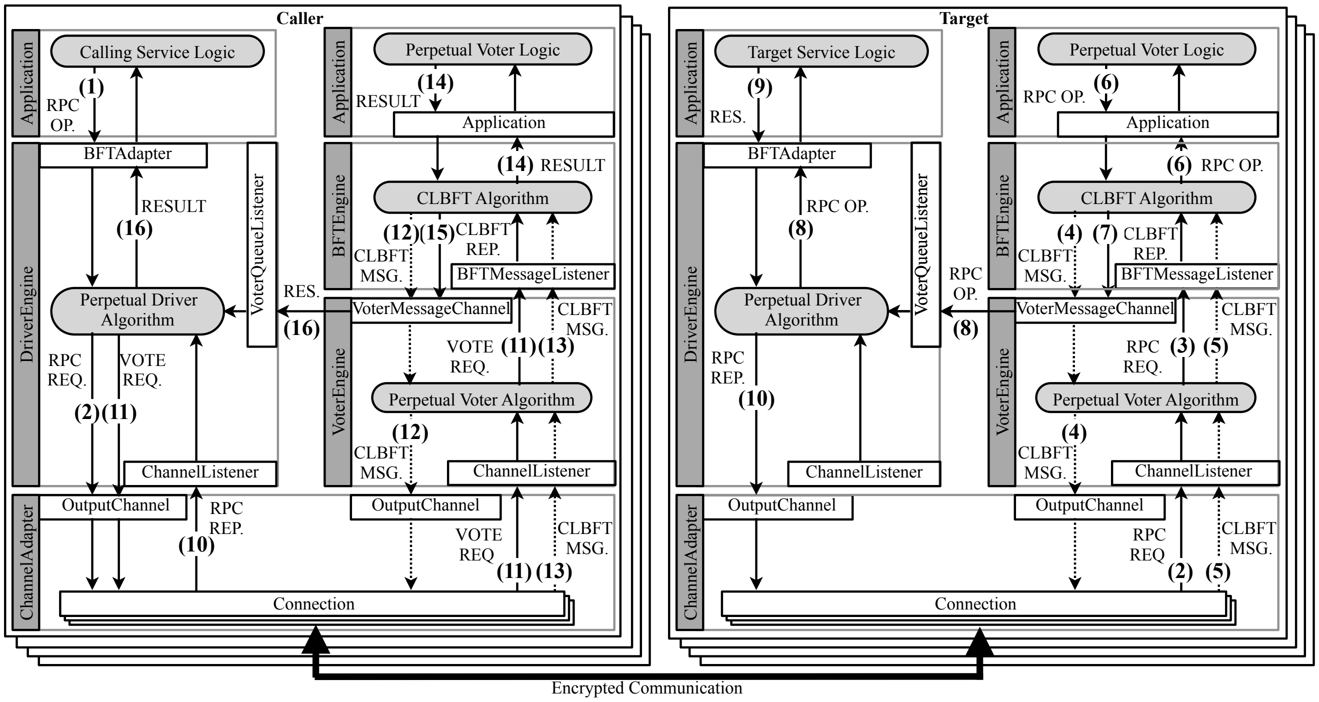

Figure 2. High-level modules (darkly shaded on the left) and interfaces (rectangles at module edges) of the Perpetual architecture. The numbered arrows show the flow of messages during fault-free execution.

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. A potential concern is replicas of c sending multiple quorums (of size $f_c + 1$) with matching requests but with different designated responders. A faulty colluding voter primary at t could selectively forward different quorums to different replicas of t. In this case, replicas of t may forward replies to different responders potentially preventing any responder from collecting the $f_t + 1$ matching replies needed for forwarding the reply to c. This does not affect correctness, though, since c is compromised to begin with.

4.2.2 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 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. Therefore, at some point after sending a request to t, each replica of c may choose to abort the request by sending an *abort request* to the voter primary of c. If at least $f_c + 1$ abort requests are received by the voter primary of c before $f_c + 1$ calling drivers forward matching replies, then the abort request may be voted upon and placed in the event queue instead of a reply from t.

4.3 Garbage Collection and Checkpoints

The state of replicas must be periodically checkpointed to persistent storage to enable them to recover from crashes. Also note that the algorithm as presented assumes unlimited space. We leverage the checkpointing mechanism to perform garbage collection and bound the amount of space used. A detailed description of both mechanisms can be found in [23].

5 System Architecture

This section describes the Perpetual system architecture. We first present the motivations for our design. We then describe the role that each major module plays in servicing an operation request.

5.1 Design Drivers

Correctness: We have formally modeled our algorithm and reasoned about its correctness using I/O Automata [23]. Our architecture mirrors the formal model closely and pays careful attention to the invariants established in the proof sketch.

Separation of Concerns: We separate high-level algorithmic functions (e.g., message processing, quorum management) from the low-level modules comprising their execution environment (e.g, network connections, authentication, and encryption)

Modularity: We employ a modular design that allows logically separate parts of the algorithm to be cleanly replaced. In particular, our implementation is not bound to specific communication or cryptographic technologies.

5.2 Modules and Interfaces

We now describe the modules and interfaces of the Perpetual architecture by tracing the execution of a request during fault-free execution, as shown in Figure 2. In stage **(1)**, the Calling Service passes a Remote Procedure Call (RPC) Operation invocation to the DriverEngine module using the BFTAdapter interface. The Perpetual Driver Algorithm augments the RPC message with the designated responder ID and a Request ID, creating a RPC Request message. In stage **(2)**, the RPC Request message is passed to the ChannelAdapter module using the OutputChannel interface. The ChannelAdapter maintains networking information within separate Connection modules. It adds authentication data (see Section 7.3) and sends the resulting message to the voter of the target primary.

The ChannelAdapter at the target primary receives and authenticates the message and passes the enclosed RPC 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 RPC Requests before sending the RPC Request to the BFTEngine module using the BFTMessageListener interface, in stage **(3)**. Stages **(4)** and **(5)** correspond to the execution of the CLBFT algorithm. Once CLBFT agreement has been reached on the RPC Request, the BFTEngine passes the RPC Operation to the Application module through the Application interface in stage **(6)**. The Perpetual voter logic (simple identity function) returns the RPC Operation back to the CLBFTEngine. In stage **(7)**, the CLBFT algorithm wraps the RPC Operation in a CLBFT Reply and passes it to the VoterEngine through the VoterMessageChannel interface. The CLBFT Reply is intercepted and the RPC Operation contained within it is extracted and forwarded to the DriverEngine using the VoterQueueListener interface, in stage **(8)**. The Target Service contained within the Application module at the target driver fetches the RPC Operation through the BFTAdapter interface and executes it. In stage **(9)**, the Result is sent back to the DriverEngine where it is encapsulated within a RPC Reply message and sent to the ChannelAdapter through the OutputChannel interface in stage **(10)** to be sent (possibly via a designated responder) to the drivers of the caller.

The ChannelAdapter of each calling driver receives the RPC Reply and passes it up to the DriverEngine module. The DriverEngine collects at least $f_t + 1$ (where t is the target service) matching RPC Replies before extracting the Result and constructing a Vote Request message. In stage **(11)**, that message is sent to the voter of the calling primary using the ChannelAdapter. Stages **(11)** through **(16)** mirror stages **(2)** through **(8)** exactly, and in stage **(16)**, the Result is consumed by the Calling Service.

6 Perpetual API

The Perpetual API, shown in Figure 3, is designed to provide a natural and flexible interface for software development that hides the replication from the application developer.

A service invokes operations on remote services using the `invoke()` method of the BFTInvoker interface. The service is required to provide a request's payload as a buffer of bytes. The method returns an `Invocation` object, representing the pending call.

Invocation objects implement Java's `Future` interface, parameterized by `Result`. The caller may call `get()` on the invocation at a later time to obtain the results of the operation. If the result is not yet available, the call blocks until it arrives. If the operation is aborted, an `OperationAborted` exception will be thrown. The method `invokeAndWait()` implements synchronous calls by invoking an operation, immediately calling `get()` on the invocation, and returning the result. The non-blocking `isCancelled()` and `isDone()` methods of `Future` throw `UnsupportedOperationException`s. Their semantics are not compatible with deterministic execution, since an invocation may complete at different times at different replicas.

An application may request a pending operation to be aborted by calling `cancel()` on its 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. Alternatively, applications may use the `setAbortPolicy()` method to specify an *abort policy* object embodying a policy for when to abort requests, replacing the default policy of waiting indefinitely for replies (see Section 4.2). Abort policies may be specified globally, per service, or per request, using variants of `invoke()` and `setAbortPolicy()` not shown. Although an abort policy may make its decisions non-deterministically (e.g. by consulting the local clock) the voter group is used to ensure that an operation is aborted deterministically and consistently on all calling replicas. It is important to note that calling `cancel()` only requests the abort of an invocation, the ultimate fate of an invocation is determined by agreement within the voter group. Hence, it is necessary to call `get()` on every invocation to learn the outcome.

A caller may request the next available result for *any* pending request using the `getNextResult()` method, making the invocation fully asynchronous. The method returns the next available reply from the event queue, block-

```
interface BFTAdapter:
  Invocation invoke(ID target, Operation op);                    // Sends the request without blocking.
  Result invokeAndWait(ID target, Operation op);             // Sends the request and waits for a reply.
  Invocation getNextResult();                      // Returns the next reply, blocking if none are available.
  void setAbortPolicy(abortPolicy policy);                                   // Sets the abort policy.
  Operation getNextRequest();                    // Returns the next request, blocking if none are available.
  void sendResult(ID caller, Result res);                            // Asynchronously sends the reply.
interface Invocation:                                      // Implements java.util.concurrent.Future.
  Result get();                                                   // Blocks if the result is not available.
  Result get(long timeOut, TimeUnit unit);     // Waits for the specified time and attempts to abort the request.
  void cancel();                                                 // Attempts to abort the pending request.
```

Figure 3. The Perpetual API provides methods to send and receive requests and replies.

ing, if necessary, until some result is available. Note that calling the `get()` method of an invocation implicitly removes that invocation from the queue.

A service that accepts incoming requests may use the `getNextRequest()` method to obtain the next pending external request. Once the external request has been executed, the service can use the `sendResult()` method to send the result back to the caller.

We omit the details of the `CheckpointListener` interface that the service must implement in order to divulge its current state for checkpointing purposes.

7 Implementation

This section details the organization of the Perpetual libraries as well as performance-related aspects of the Perpetual implementation. Perpetual is implemented completely in Java. The code base is organized into three libraries, containing the CLBFT code, the Perpetual code, and the Communications layer code.

7.1 The CLBFT Library

Our initial strategy was to use the BASE [24] implementation of CLBFT, by giving the C++ BASE library a Java Native Interface (JNI) wrapper. However, the BASE implementation is not modular, tightly coupling the high-level CLBFT algorithm with low-level (connections, cryptography) concerns[5]. Therefore, we implemented our own version of CLBFT in Java, which abstracts from cryptography and network functions through interfaces. The CLBFT library contains the BFTEngine module.

7.2 The Perpetual Core Library

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

7.3 The Communications Library

DriverEngines and VoterEngines communicate via the ChannelAdapter (CA) module, which provides a high-level connectionless group-oriented messaging interface. The CA, contained in the Communications Library, encapsulates all networking, authentication, and encryption functions. It uses Java's JCE framework, permitting easy substitution of cryptography provider libraries. The current CA implementation uses SSL/TLS [25] over TCP/IP. We favor flow- and congestion-controlled TCP over connectionless transports such as UDP, to best support geographically dispersed replica groups. However, the CA's architecture allows different transport protocols to be plugged in. Although designed for Perpetual, the CA is general enough to be useful for other systems needing secure group communication.

The CA guarantees exactly-once (possibly out-of-order) delivery of messages between correct replicas, as required by Perpetual. The TCP/SSL transport assigns 64-bit sequence numbers to messages and uses explicit message acknowledgements to ensure delivery, even when connections break and are re-established. Acknowledgements are inserted into the data stream every n messages, efficiently encoded as 16-bit integers indicating the number of messages received since the last acknowledgement.

The CA guarantees the authenticity and privacy of messages. Replicas are identified with RSA public keys, which are used to establish SSL sessions. Direct messages between pairs of replicas need no further authentication, but the CA must also guarantee authenticity of messages that are sent indirectly via forwarding through (possibly malicious) intermediary replicas. To this end, a CA includes with each forwardable message a MAC authenticator [19] for each end-destination recipient replica, preventing the intermediary replica from altering the message before forwarding it. A correct intermediary CA collects the MACs from all incoming copies of a message and then sends to each end-recipient the MACs intended for it (i.e., one MAC from each of the original sending replicas.) The relaying call includes all MACs from all replicas, to enable the relay receiver to forward them on.

7.4 Optimizations

Our implementation is designed to efficiently scale to thousands of concurrent connections. It uses the Java `NIO` asynchronous I/O library, worker thread

[5]BASE also needs cryptography routines from the discontinued SFS [20] project

pooling through the `ExecutorService` framework, and pooling of `ByteBuffers` using wait-free `ConcurrentLinkedQueues`.

We strive to minimize data copying as messages are passed through different modules by utilizing the "gather" calls of `SocketChannel` and `SSLEngine`. This allows the application layer, the Perpetual layer and CA layer to provide their respective message headers without additional copying.

8 Experimental Results and Analysis

The main purpose of our experiments was to show that Perpetual is scalable and efficient. We began by measuring operation throughput as the number of calling and target replicas was varied, using groups of size 1, 4, 7, and 10, tolerating $f = 0, 1, 2$, and 3 Byzantine faulty replicas, respectively. We then measured the performance as network latency was varied to simulate the behavior of Perpetual middleware in both Local Area Network (LAN) and Wide Area Network (WAN) settings. Experimental results for related work in the field have only shown the performance on LANs to the best of our knowledge. However, achieving true replica failure independence requires replicas to be hosted in geographically dispersed locations and subnetworks. To validate our simulated latency results, we repeated a subset of our experiments on the PlanetLab [8] research network. Finally, we performed experiments to evaluate the effects of non-zero execution time and bandwidth load on Perpetual, as well as evaluating the performance gains made possible by Perpetual's support for asynchronous parallel operations.

Our LAN experiments were performed on a dedicated Washington University testbed [26] made up of 2GHz Opteron machines with 512 MB of RAM, connected via a Netgear GSM7352S Gigabit Ethernet router (with the `ping` tool reporting $85\mu s$ pairwise RTTs). All machines ran RedHat Desktop 4 (kernel version 2.6.9-42.0.3.EL). The WAN experiments were carried out on PlanetLab nodes running Fedora Core 4 (kernel version 2.6.12-1.1398_FC4) with our slice reserved using the Sirius Calendar. We used two groups of PlanetLab nodes, chosen to have as similar pairwise latencies as possible, using data from the S^3 Scalable Sensing Service [27]. The nodes in the "fast group" had an average pair-wise latency of 23.52ms, whereas the "slow group" had an average pair-wise latency of 41.68ms, as measured by the replicas themselves during runs. All tests used Java Runtime version 1.6.0_01. The only additional run-time parameter was to cap the heap size at 128MB. The SSL ciphersuite used was RSA/RC4/MD5.

8.1 Replica scalability

In our first set of experiments we varied the number of calling and target replicas executing a particular test. In all our tests (save for the asynchronous test in Section 8.4) the calling service invokes an operation that increments an integer value stored in the target service by a parameter-specified value. The target computes the new value, stores it in the target's state and sends the new value in its reply to the caller. The caller invokes the operation to increment the value one thousand times in sequence, invoking the operation again as soon as it has received the reply to the previous invocation. It then sends one thousand requests to decrement the value, verifying at the end of the test that the target is back to the original value.

Figure 4 shows the throughput in operations per second on the vertical axis, while running on the LAN, computed as the number of operations executed divided by the total time to complete the test. The horizontal axis shows the different calling group sizes and the bars correspond to different sizes of target groups. Figure 5 shows the same test running on PlanetLab wide area "fast" nodes.

Observe that the first bar, showing the case of a single target replica and single calling replica, corresponds to a baseline case of a non-fault-tolerant caller calling a non-fault-tolerant target. A singleton group in our implementation does not go through any rounds of Perpetual protocol messages but processes requests and replies immediately, so only the overhead of authentication and encryption is incurred. The other bars in the first group, with a singleton calling group and replicated targets, correspond to the baseline case of a non-fault-tolerant caller calling a replicated passive CLBFT service.

On the LAN, the overhead of CLBFT replication over no replication is 213%. The overhead of Perpetual replication over CLBFT(which does not replicate the caller), however, is 62%, 89% and, 95%, in the cases of 4, 7, and 10 calling and target replicas, respectively. The wide-area results paint a similar picture, validating our testbed results. The pairwise latencies varied greatly on the WAN, with a standard deviation of 28.7ms, so the bars involving only a few replicas are not highly significant. The first bar, for $n_s = n_c = 1$

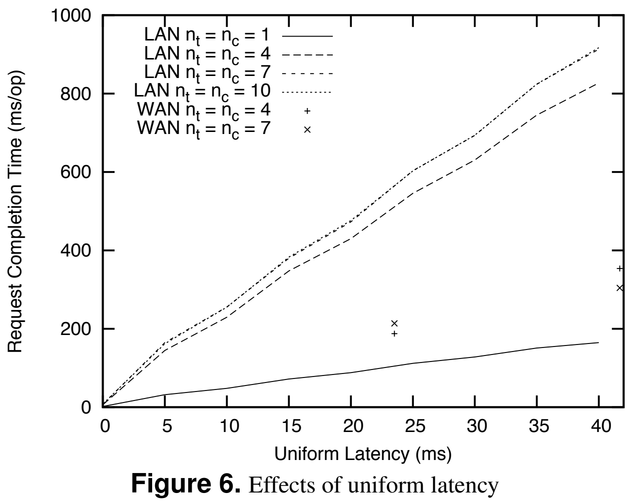

Figure 6. Effects of uniform latency

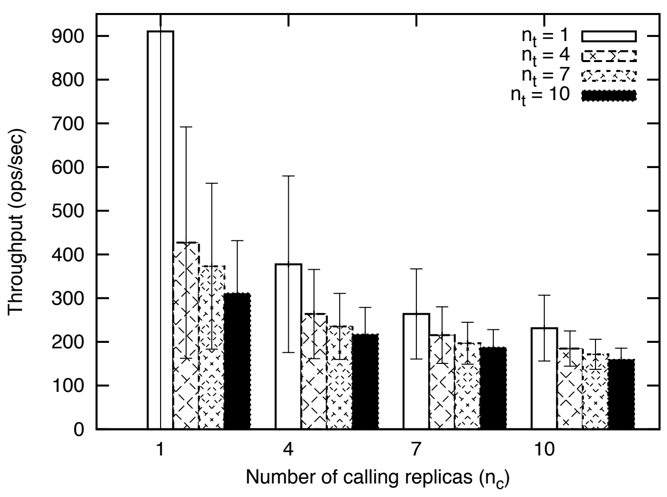

Figure 4. LAN replica scalability

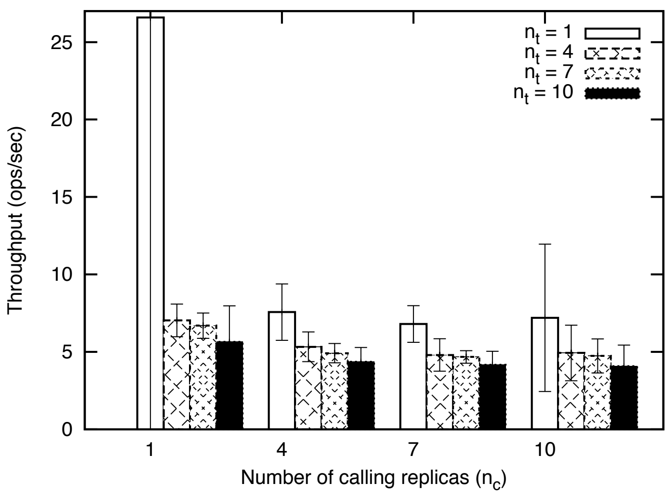

Figure 5. WAN replica scalability

is essentially arbitrary, since different pairs of replicas could have achieved a significantly higher or lower throughput.

The graph confirms what we would expect from Perpetual, which has a constant number of message hops per request. It scales well to high degrees of replication, with minimal decreases in throughput as group sizes increase. Note, though, that these results represent the worst-case overhead, since operations execute almost instantly. As the graphs in Section 8.4 demonstrate, the relative overhead is much less with non-trivial operations.

8.2 Effects of latency

We ran experiments designed to measure the effects of latency on the performance of Perpetual replica groups. In our first set of tests, we uniformly varied the latency between all pairs of replicas by testing on the LAN but artificially delaying the sending of messages. Figure 6 plots average operation execution times (elapsed time divided by number of operations) for selected latencies and group sizes. Increases in latency have a much greater effect on the replicated cases than on the unreplicated case, which does not go through rounds of agreement. The size of replica groups, however, does not make a big difference. In fact, the lines for $n = 7$ and $n = 10$ coincide. The graph includes our data points from our PlanetLab WAN experiments, for the cases of $n = 4$ and $n = 7$ nodes from the "fast" and "slow" groups. The higher performance is explained by the high variance of pair-wise latencies and the fact that the fastest $2f + 1$ replicas effectively dictate the performance.

8.3 Effects of load

For the experiments in sections 8.1 through 8.2, the operations were essentially null-operations requiring small amounts of CPU time and network bandwidth. In the next set of experiments, we investigate the performance and overhead of Perpetual as CPU load and bandwidth use are varied.

We used the same test as before but with additional processing time and/or delay for each operation. Figure 7 plots average request completion times for various CPU loads, with groups of 4, 7, and 10 replicas. We created loads by computing SHA-1 hashes of random data buffers of varying sizes, calibrating their size as to achieve our desired operation processing times. The graph also shows the relative overhead of Perpetual replication, defined as replica group operation completion time divided by non-replicated completion time. Figure 9 shows the same test but this time half of the processing time is created through hash computations while the other half is created by letting the thread sleep, simulating disk I/O time.

The request completion time is linear in operation execution time, but note that the relative overhead of using BFT rapidly diminishes as operation execution time increases. For example, with a 6ms mixed operation roughly corresponding to a disk access and some processing, Perpetual replication with $n_t = n_c = 4$ adds around 25% of overhead over the case with no replication.

Figure 8 plots average completion times for various sizes of operation requests. Requests were inflated by padding them with random data. The completion times grow linearly with request sizes, up until the point where network processing becomes the bottleneck.

8.4 Effects of asynchronous invocation

For the experiments in sections 8.1 through we used synchronous requests, where a caller waits for the reply to each request before issuing a new one. We also performed an experiment to gauge performance when issuing asynchronous requests. We modified the experiment so that instead of issuing one request at a time, callers would issue a batch of requests and then wait for all of them to complete, before

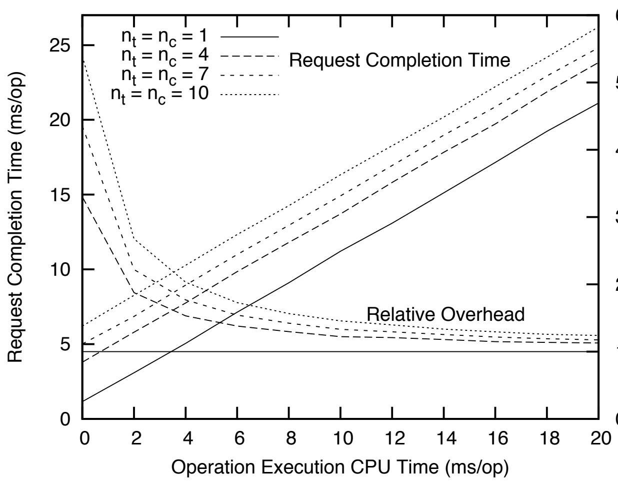

Figure 7. Effects of CPU load

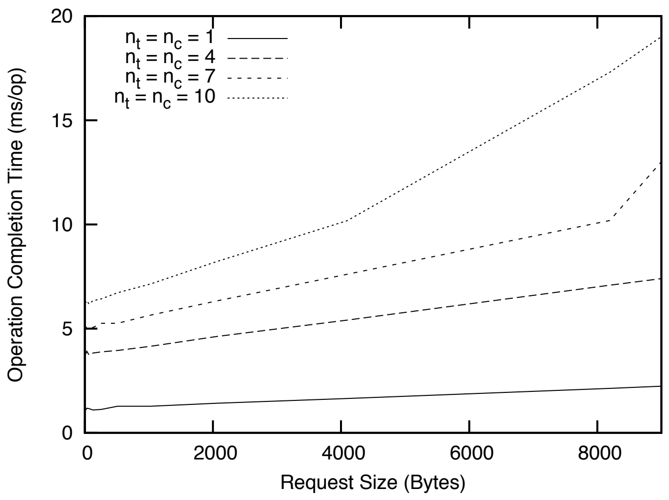

Figure 8. Effects of request size

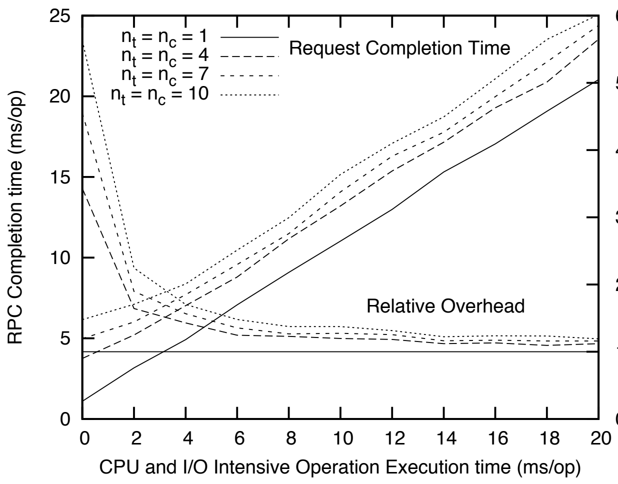

Figure 9. Effects of mixed load

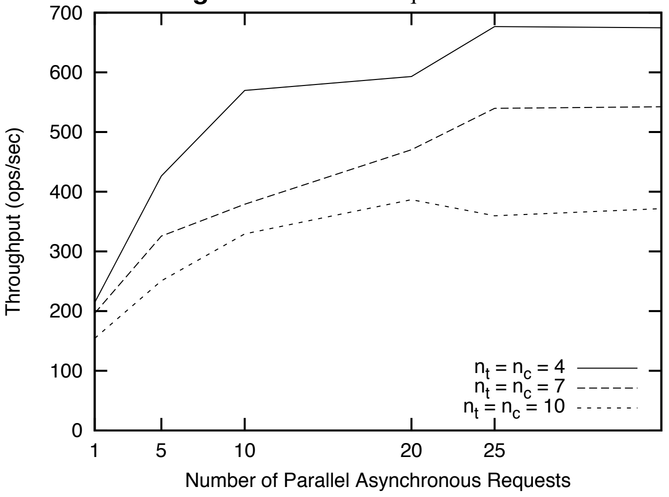

Figure 10. Effects of asynchronous requests

issuing the next batch. The size of parallel request batches was varied from 1 to 50. As can be seen in Figure 10, this results in a significant increase in throughput. In the case of $n = 4$, the improvement is more than threefold, which bodes well for applications that can take advantage of concurrent requests. The benefits begin to taper off around 25 parallel requests. We believe the replicas become compute-bound at that point but intend to investigate this more thoroughly in the future.

9 Conclusion

We have presented the algorithm, implementation, and performance evaluations of Perpetual, middleware that supports correct execution of complex deterministic services in spite of Byzantine faults. The results demonstrate that Perpetual has moderate overhead for non-trivial operations, and it scales well to large replica groups. Perpetual's support for interoperability of services with differing degrees of fault-tolerance and all the major application communication models paves the way for deployment of Byzantine fault tolerant replication of critical distributed applications.

We are currently working toward providing Byzantine fault tolerance to existing middleware technologies such as SOAP [28] and CORBA [21]. Although some work has been done in this field [15, 18, 16], our approach will be uniquely suited for developing practical systems that require interaction between replicated services.

10 Acknowledgements

We thank Charlie Wiseman, Ken Wong, and the rest of the ONL team for their support in using ONL. We also thank Michael Groner, Bob Lozano, and Justin Honold at Appistry Inc. for the use of their testbed as well as Sergey Gorinsky for his assistance with PlanetLab. The researchers were supported in part by National Science Foundation grants 0305954, 0618266, and 0722328.

References

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

[2] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter. Efficient Byzantine-Tolerant Erasure-Coded Storage. In *Proc. 5th Intl. Conf. on Dependable Systems and Networks*, pages 135–144, 2004.

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

[4] Google Inc. *Google Maps API Concepts*, October 2007.

[5] Google Inc. *Google Checkout XML API Developer's Guide*, 2007.

[6] Amazon Web Services. *Amazon Flexible Payments Service*, January 2007.

[7] 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*, 2006.

[8] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33(3):3–12, 2003.

[9] Computer Emergency Response Team Coordination Center (CERT/CC). Statistics 1988-2006, `http://www.cert.org`.

[10] Symantec corporation. Symantec internet security threat report, volume x, September 2006.

[11] A.G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM Systems Journal*, 42(1):5–18, 2003.

[12] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4d approach to network control and management. *SIGCOMM Comput. Commun. Rev.*, 35(5), 2005.

[13] Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. Self-organising software architectures for distributed systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 33–38, 2002.

[14] 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.

[15] P. Narasimhan, K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Providing Support for Survivable CORBA Applications with the Immune System. In *Proc. 19th Intl. Conf. on Distributed Computing Systems*, pages 507–516, 1999.

[16] 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.

[17] S. Ahmed. A Scalable Byzantine Fault Tolerant Secure Domain Name System, 2001. Master's thesis, Massachusetts Institute of Technology.

[18] M. G. Merideth, A. Iyengar, T. Mikalsen, S. Tai, I. Rouvellou, and P. Narasimhan. Thema: Byzantine-Fault-Tolerant Middleware forWeb-Service Applications. In *Proc. 24th Symp. on Reliable Distributed Systems*, pages 131–140, 2005.

[19] 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.

[20] D. Maziores, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 124–139, New York, NY, USA, 1999. ACM Press.

[21] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, 2.5 edition, September 2001.

[22] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC '87: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 137–151, New York, NY, USA, 1987. ACM Press.

[23] I. Wehrman, S. L. Pallemulle, and K. J. Goldman. Extending Byzantine Fault Tolerance to Replicated Clients. Technical Report WUCSE-2006-7, Washington University, 2006.

[24] 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.

[25] T. Dierks. RFC 4346: the Transport Layer Security (TLS) Protocol Version 1.1, 2006.

[26] J. DeHart, F. Kuhns, J. Parwatikar, J. Turner, C. Wiseman, and K. Wong. The open network laboratory. *SIGCSE Proceedings*, Mar 2006.

[27] P. Yalagandula, P. Sharma, S. Banerjee, S. Basu, and S. Lee. S3: a scalable sensing service for monitoring large networked systems. In *INM '06: 2006 SIGCOMM workshop on Internet network management*, pages 71–76, New York, NY, USA, 2006. ACM Press.

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