Appeared in ***7th USENIX Symposium on Network Design and Implementation (NSDI '10)***

Prophecy: Using History for High-Throughput Fault Tolerance

Siddhartha Sen, Wyatt Lloyd, and Michael J. Freedman
Princeton University

Abstract

Byzantine fault-tolerant (BFT) replication has enjoyed a series of performance improvements, but remains costly due to its replicated work. We eliminate this cost for read-mostly workloads through Prophecy, a system that interposes itself between clients and any replicated service. At Prophecy's core is a trusted *sketcher* component, designed to extend the semi-trusted load balancer that mediates access to an Internet service. The sketcher performs fast, load-balanced reads when results are historically consistent, and slow, replicated reads otherwise. Despite its simplicity, Prophecy provides a new form of consistency called *delay-once consistency*. Along the way, we derive a distributed variant of Prophecy that achieves the same consistency but without any trusted components.

A prototype implementation demonstrates Prophecy's high throughput compared to BFT systems. We also describe and evaluate Prophecy's ability to scale-out to support large replica groups or multiple replica groups. As Prophecy is most effective when state updates are rare, we finally present a measurement study of popular websites that demonstrates a large proportion of static data.

1 Introduction

Replication techniques are now the norm in large-scale Internet services, in order to achieve both reliability and scalability. However, leveraging active agreement to *mask* failures, whether to handle fail-stop behavior [41, 50] or fully malicious (Byzantine) failures [42], is not yet widely used. There is some movement in this direction from industry—such as Google's Chubby [10] and Yahoo!'s Zookeeper [66] coordination services, based on Paxos [41]—but both are used to manage infrastructure, not directly mask failures in customer-facing services.

And yet non-fail-stop failures in customer-facing services continue to occur, much to the chagrin and concern of system operators. Failures may arise from malicious break-ins, but they also may occur simply from system misconfigurations: Facebook leaking source code due to *one* misconfigured server [60], or Flickr mixing up returned images due to *one* improper cache server [24]. In fact, both of these examples could have been prevented through redundancy and agreement, without requiring full N-version programming [8]. The perceived need for systems robust to Byzantine faults—a superset of misconfigurations and Heisenbugs—has spurned almost a cottage industry on improving performance results of Byzantine fault tolerant (BFT) algorithms [1, 6, 12, 17, 30, 37, 38, 56, 62, 64, 65, 67].

While the latency of recent BFT algorithms has approached that of unreplicated reads to individual servers [15, 38, 64], the throughput of such systems falls far short. This is simple math: a minimum of four replicas [12] (or sometimes even six [1]) are required to tolerate one faulty replica, and at least three must participate in each operation. For datacenters in the (tens of) thousands of servers, requiring four times as many servers for the same throughput may be a non-starter. Even services that already replicate their data, such as the Google File System [25], would see their throughput drop significantly when using BFT agreement.

But if the replication cost of BFT is provably necessary [9], something has to give. One might view our work as a thought experiment that explores the potential benefit of placing a small amount of trusted software or hardware in front of a replicated service. After all, wide-area client access to an Internet service is typically mediated by some middlebox, which is then at least trusted to provide access to the service. Further, a small and simple trusted component may be less vulnerable to problems such as misconfigurations or Heisenbugs. And by treating the back-end service as an abstract entity that exposes a limited interface, this simple device may be able to interact with both complex and varied services. Our implementation of such a device has less than 3000 lines of code.

Barring such a solution, most system designers opt either for cheaper techniques (to avoid the costs of state machine replication) or more flexible techniques (to ensure service availability under heavy failures or partitions). The design philosophies of Amazon's Dynamo [18], GFS [25], and other systems [20, 23, 61] embrace this perspective, providing only eventually-consistent storage. On the other hand, the tension between these competing goals persists, with some systems in industry re-introducing stronger consistency properties. Examples include timeline consistency in Yahoo!'s PNUTS [16] and per-user cache invalidation on Facebook [21]. Nevertheless, we are unaware of any major use of *agreement* at the front-tier of customer-facing services. In this paper, we challenge the assumption that the tradeoff between strong consistency and cost in these services is fundamental.

This paper presents Prophecy, a system that lowers the performance overhead of fault-tolerant agreement for customer-facing Internet services, at the cost of slightly weakening its consistency guarantees. At Prophecy's core is a trusted *sketcher* component that mediates client access to a service replica group. The sketcher maintains a compact history table of observed request/response pairs; this history allows it to perform fast, load-balanced reads when state transitions do not occur (that is, when the current response is identical to that seen in the past) and slow, replicated reads otherwise (when agreement is required). The sketcher is a flexible abstraction that can *interface with any replica group*, provided it exposes a limited set of defined functionality. This paper, however, largely discusses Prophecy's use with BFT replica groups. Our contributions include the following:

- When used with BFT replica groups that guarantee linearizability [32], Prophecy significantly increases throughput through its use of fast, load-balanced reads. However, it relaxes the consistency properties to what we term *delay-once* semantics.
- We also derive a distributed variant of Prophecy, called D-Prophecy, that similarly improves the throughput of traditional fault-tolerant systems. D-Prophecy achieves the same delay-once consistency but *without any trusted components*.
- We introduce the notion of *delay-once consistency* and define it formally. Intuitively, it implies that faulty nodes can at worst return only stale (not arbitrary) data.
- We demonstrate how to scale-out Prophecy to support large replica groups or many replica groups.
- We implement Prophecy and apply it to BFT replica groups. We evaluate its performance on realistic workloads, not just null workloads as typically done in the literature. Prophecy adds negligible latency compared to standard load balancing, while it provides an almost linear-fold increase in throughput.
- Prophecy is most effective in read-mostly workloads where state transitions are rare. We conduct a measurement study of the Alexa top-25 websites and show that over 90% of requests are for mostly static data. We also characterize the dynamism in the data.

Table 1 summarizes the different properties of a traditional BFT system, D-Prophecy, and Prophecy. The remainder of this paper is organized as follows. In §2 we motivate the design of D-Prophecy and Prophecy, and we describe this design in §3. In §4 we define delay-once consistency and analyze Prophecy's implementation of this consistency model. In §5 we discuss extensions to the basic system model that consider scale and complex component topologies. We detail our prototype implementation in §6 and describe our system evaluation in §7. In §8 we present our measurement study. We review related work in §9 and conclude in §10.

Property	BFT	D-Prophecy	Prophecy
Trusted components	No	No	Yes
Modified clients	Yes	Yes	No
Session length	Long	Long	Short, long
Load-balanced reads	No	Yes	Yes
Consistency	Linearized	Delay-once	Delay-once

Table 1: **Comparison of a traditional BFT system, D-Prophecy, and Prophecy.**

2 Motivating Prophecy's Design

One might rightfully ask whether Prophecy makes unfair claims, given that it achieves performance and scalability gains at the cost of additional trust assumptions compared to traditional fault-tolerant systems. This section motivates our design through the lens of BFT systems, in two steps. First, we improve the performance of BFT systems on realistic workloads by introducing a cache at each replica server. By optimizing the use of this cache, we derive a distributed variant of Prophecy that does not rely on any trusted components. Then, we apply this design to customer-facing Internet services, and show that the constraints of these services are best met by a shared, trusted cache that proxies client access to the service replica group. The resulting system is Prophecy.

In our discussion, we differentiate between *write requests*, or those that modify service state, and *read requests*, or those that simply access state.

2.1 Traditional BFT Services and Real Workloads

A common pitfall of BFT systems is that they are evaluated on null workloads. Not only are these workloads unrealistic, but they also misrepresent the performance overheads of the system. Our evaluation in §7 shows that the cost of executing a non-null read request in the PBFT system [12] dominates the cost of agreeing on the ordering of the request, even when the request is served entirely from main memory. Thus the PBFT read optimization, which optimistically avoids agreement on read requests, offers little or no benefit for most realistic workloads. Improving the performance of read requests requires optimizing the *execution* of the reads themselves.

Unlike write requests, which modify service state and hence must be executed at each replica server, read requests can benefit from causality tracking. For example, if there are no causally-dependent writes between two identical reads, a replica server could simply cache the response of the first read and avoid the second read altogether.[1] However, this requires (1) knowledge of the causal dependencies of all write requests, and (2) a response cache of all prior reads at each replica server. The first requirement is unrealistic for many applications: a single write may modify the service state in complex ways. Even if we address this problem by invalidating the entire response cache upon receiving any write, the space needed by such a cache could be prohibitive: a cache of Facebook's 60+ billion images on April 30, 2009 [49], assuming a scant 1% working-set size, would occupy approximately 15TB of memory. Thus, the second requirement is also unrealistic.

Instead of caching each response r, the replica servers can store a compact, collision-resistant sketch $s(r)$ to enable *cache validation*. That is, when a client issues a read request for r, only one replica server executes the read and replies with r, while the remaining replica servers reply with $s(r)$ from their caches. The client accepts r only if the replica group agrees on $s(r)$ and if $s(r)$ validates r. Thus, even if the replica that returns r is faulty, it cannot make the client accept arbitrary data; in the worst case, it causes the client to accept a stale version of r. Therefore we only need to ask one replica to execute the read, effectively implementing what we call a *fast read*. Fast reads drastically improve the throughput of read requests and can be load-balanced across the replica group to avoid repeated stale results. The replica servers maintain a fresh cache by updating it during regular (replicated) reads, which are issued when fast reads fail. Using a compact cache reduces the memory footprint of the Facebook image working set to less than 27GB.

We call the resulting system Distributed Prophecy, or D-Prophecy, and call the consistency semantics it provides *delay-once consistency*.

2.2 BFT Internet Services

An oft-overlooked issue with BFT systems, including D-Prophecy, is that they are *implicitly* designed for services with long-running sessions between clients and replica servers (or at least always presented and evaluated as such). Clients establish symmetric session keys with each replica server, although the overhead of doing so is not typically included when calculating system performance. Figure 1 shows the throughput of the PBFT implementation as a function of session length, with all relevant optimizations enabled including the read optimization (indicated by 'ro'). As sessions get shorter, throughput is drastically reduced because replicas need to decrypt and verify clients' new session keys. For PBFT sessions consisting of 128 read requests, throughput is half of its maximum, and for sessions consisting of 8 read requests, throughput is one-tenth of its maximum.

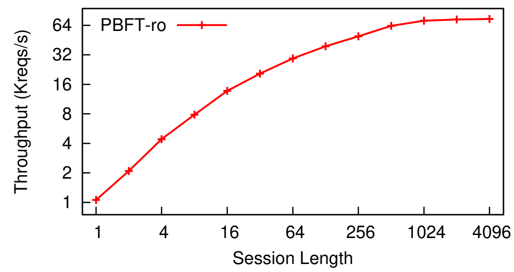

Figure 1: **PBFT's throughput in the thousands of requests per second for null requests in sessions of varying length. Note that both axes are log scale.**

The assumption of long-lived sessions breaks down for Internet services, however, which are mostly characterized by *short-lived sessions* and *unmodified clients*. These properties make it impractical for clients to establish per-session keys with each replica. Moreover, depending on clients to perform protocol-specific tasks leads to poor backwards compatibility for legacy clients of Internet services (*e.g.*, web browsers), where cryptographic support is not easily available [2]. Instead, we might turn to using an entity knowledgeable of the BFT protocol to proxy client requests to a service replica group. And since Internet services already rely on the correct operation of local middleboxes (at least with respect to service availability), we extend this reliance by converting the middlebox into a trusted proxy. The trusted proxy interfaces multiple short-lived sessions between clients and itself with a single long-lived session between itself and the replica group, acting as a client in the traditional BFT sense.

When using proxied client access to a D-Prophecy group, there is no need to maintain redundant caches at each replica server: a shared cache at the trusted proxy suffices, and it preserves delay-once consistency. A fast read now mimics the performance of an unreplicated read, as the proxy only asks one replica server for r and validates the response with its (local) copy of $s(r)$. Since the cache is compact, the proxy remains a small and simple trusted component, amenable to verification. We call this system Prophecy, and present its design in §3.

2.3 Applications

The delay-once semantics of Prophecy imply that faulty nodes can at worst return stale (not arbitrary) data. This

[1] Other causality-based optimizations, such as client-side speculation [64] or server-side concurrent execution [37] are also possible, but are complementary to any cache-based optimizations.

semantics is sufficient for a variety of applications. For example, Prophecy would be able to protect against the Facebook and Flickr mishaps mentioned in the introduction, because it would not allow arbitrary data to reach the client. Applications that serve inherently static (write-once) data are also good candidates, because here a "stale" response is as fresh as the latest response. In §8 we demonstrate the propensity for static data in today's most popular websites.

Social networks and "Web 2.0" applications are good candidates for delay-once consistency because they typically do not require all writes to be immediately visible. Consider the following example from Yahoo!'s PNUTS system [16]. A user wants to upload spring-break photos to an online photo-sharing site, but does not want his mother to see them. So, he first removes her from the permitted access list of his database record and then adds the spring-break photos to this record. A consistency model that allows these updates to appear in different orders at different replicas, such as eventual consistency [22], is insufficient: it violates the user's intention of hiding the photos from his mother. Delay-once consistency only allows stale data to be returned, not data out-of-order: if the photos are visible, then the access control update must have already taken place. Further, once the user has "refreshed" his own page and sees the photos, he is guaranteed that his friends will also see them.

For applications where writes are critical, such as a bank account, delay-once consistency is appropriate because it ensures that writes follow the protocol of the replica group. Although reads may return stale results, they can only do so in a limited way, as we discuss in §4. On the other hand, there are some applications for which delay-once consistency is not beneficial, such as those that critically depend on reading the latest data (*e.g.*, a rail signaling service), or those that return nondeterministic content (*e.g.*, a CAPTCHA generator).

3 System Design

We first define a sketcher abstraction that lies at the heart of Prophecy. For a more traditional setting, we use this sketcher to design a distributed variant of Prophecy, or D-Prophecy. We then present the design of Prophecy.

3.1 The Sketcher

Prophecy and D-Prophecy use a sketcher to improve the performance of read requests to an existing replica group. A *sketcher* maintains a history table of compact, collision-resistant sketches of requests and responses processed by a replica group. Each entry in the history table is of the form $(s(q), s(r))$, where q is a request, r is the response to q, and s is the sketching function used

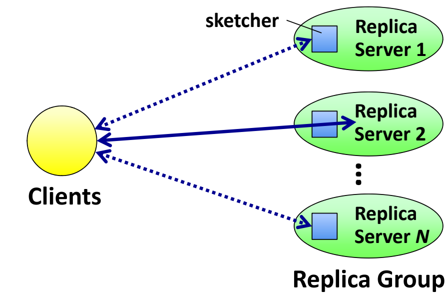

Figure 2: **Executing a fast read in D-Prophecy. Only one replica server executes the read (bold line); the others return the response sketch in the history table (dashed lines).**

for compactness (s typically makes use of a secure hash function like SHA-1). The sketcher looks up or updates entries in the history table using a standard get/set interface, keyed by $s(q)$. In Prophecy, only read requests and responses are stored in the history table.

The specific use of the sketcher and its interaction with the replica group differs between Prophecy and D-Prophecy. However, both systems require the replica group to support the following request interface:

- RESP $\leftarrow$ *fast*(REQ q)
- (RESP r, SEQ_NO σ) $\leftarrow$ *replicated*(REQ q)

We expect the *fast* interface to be new for most replica groups. The *replicated* interface should already exist, but may need to be extended to return sequence numbers. No modifications are made to the replica group beyond what is necessary to support the interfaces, in either system.

3.2 D-Prophecy

Figure 2 shows the system model of D-Prophecy. Except for the sketcher, all other entities are standard components of a replicated service: clients send requests to (and receive responses from) a service implemented by N replica servers, according to some replication protocol like PBFT. Each replica server is augmented with a sketcher that maintains a history table for read requests. The history table is read by the *fast* interface and updated by the *replicated* interface, as follows.

A client issues a fast read q by sending it to all replica servers and choosing one of them to execute q and return r. The policy for selecting a replica server is unspecified, but a uniformly random policy has especially useful properties (see §4.2). The other replicas use their sketcher to lookup the entry for $s(q)$ and return the corresponding response sketch $s(r)$, or null if the entry does not exist. If the client receives a quorum of non-null response sketches that match the sketch of the actual response, it accepts the response. The quorum size depends on the replication protocol; we give an example

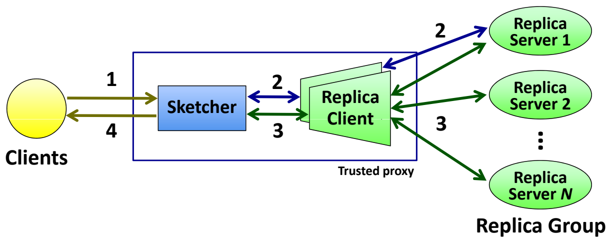

Figure 3: **Prophecy mediating access to a replica group.**

below. Otherwise, we say a *transition* has occurred and the client reissues the request as a replicated read. A replicated read is executed according to the protocol of the replica group, with one additional step: all replica servers use their sketcher to update the entry for $s(q)$ with the new value of $s(r)$, before sending a response to the client.

Readers familiar with the PBFT protocol will notice that fast reads in D-Prophecy look very similar to PBFT optimized reads. However, there is a crucial difference: PBFT requires every replica server to execute the read, while D-Prophecy requires only one such execution, performing in-memory lookups of $s(r)$ at the rest. For non-null workloads, this represents a significant performance improvement, as shown in §7. On the flip side, each replica server requires additional memory to store its history table, though in practice this overhead is small. The quorum size required for fast reads is identical to the quorum size required for optimized reads: $(2N+1)/3$ responses suffices with some caveats (see §5.1.3 of [11]), and N always suffices.

The architecture of D-Prophecy resembles that of a traditional BFT system: clients establish session keys with the replica servers and participate fully in the replication protocol. As we observed in §2.2, this makes D-Prophecy unsuitable for Internet services, with their environment of short-lived sessions and unmodified clients. This motivates the design of Prophecy, discussed next.

3.3 Prophecy

Figure 3 shows the simplest realization of Prophecy's system model. (We consider extensions to the basic model in §5.) There are four types of entities: clients, sketchers, replica clients, and replica servers. Unmodified clients' requests to a service are handled by the sketcher; together with the replica clients, this serves as the trusted proxy described in §2.2. The replica clients interact with the service, implemented by a group of N replica servers, according to some replication protocol.

The sketcher issues each request through a replica client; the next subsection details the handling of requests. Functionally, the sketcher in Prophecy plays the same role as the per-replica-server sketchers in D-Prophecy. Architecturally, however, its role is quite different. In Prophecy, a fast read is sent only to the single replica server that executes it, and neither the *fast* nor *replicated* interface accesses the history table directly. Thus, the replica group is treated as a black box. Since the sketcher is external to the replica group, writes processed by the group may no longer be visible or discernible to the sketcher; *i.e.*, there may exist an *external write channel*. Since only replica clients interact directly with the replica servers, each replica client can maintain a single, long-lived session with each replica server. Wide-area clients are shielded from any churn in the replica group and are unaware of the replication protocol: the only responses they see are those that have already been accepted by the sketcher.

The type of session used between clients and the sketcher is left open by our design, as it may vary from service to service. For example, services that only allow read or simple write operations (*e.g.*, HTTP GETs and POSTs) may use unauthenticated sessions. A service like Facebook may use authentication only during user login, and use unauthenticated cookie-based sessions after that. Finally, services that store private or protected data, such as an online banking system, may secure sessions at the application level (*e.g.*, using HTTPS). Prophecy's architecture makes it easy to cope with the overhead of client-sketcher authentication, because one can simply add more sketchers if this overhead grows too high (see §5). To achieve the same scale-out effect, traditional BFT systems like PBFT and D-Prophecy would need to add entire replica groups.

3.3.1 Handling a Request

The sketcher stores two additional fields with each entry $(s(q), s(r))$ in the history table: the sequence number σ associated with r, and a 2-bit value b indicating whether $s(q)$ is *whitelisted* (always issued as a fast read), *blacklisted* (always issued as a replicated request), or neither (the default). The sketch $s(r)$ is empty for whitelisted or blacklisted requests. Algorithm 1 describes the processing of a request and is illustrated in Figure 3 (numbers on the right correspond to the numbered steps in the figure).

Prophecy requires a sequence number to be returned by *replicated*, as it seeks to issue concurrent requests to the replica group using multiple replica clients. Concurrency allows reads to execute in parallel to improve throughput. Unfortunately, a sketcher that issues requests concurrently has no way of discerning the correct order of replicated reads by itself, *i.e.*, the order they were processed by the replica group. Thus, it relies on the sequence number returned by *replicated* to ensure that entries in the history table always reflect the latest system state.

The sketcher requires some application-specific knowledge of the format of q and r. This information is used

Algorithm 1 Processing a request at the sketcher.

Receive request q from client (1)
if q is a read request **then**
  $(s(q), s(r), \sigma, b) \leftarrow$ Lookup $s(q)$ in history table
  if $(s(r) \neq \text{null})$ **and** $(b \neq \text{blacklisted})$ **then**
    $r' \leftarrow \textit{fast}(q)$ (2)
    if $(s(r') = s(r))$ **or** $(b = \text{whitelisted})$ **then**
      return r' to client (4)
    end if
  end if
  $(r', \sigma') \leftarrow \textit{replicated}(q)$ (3)
  if $(s(r) = \text{null})$ **or** $(\sigma' > \sigma)$ **then**
    Update history table with $(s(q), s(r'), \sigma', b)$
  end if
else
  $(r', \sigma') \leftarrow \textit{replicated}(q)$ (3)
end if
return r' to client (4)

to determine if q is a read or write request, and to discard extraneous or non-deterministic information from q or r while computing $s(q)$ or $s(r)$. For example, in our prototype implementation of Prophecy, an HTTP request is parsed by an HTTP protocol handler to extract the URL and HTTP method of the request; the same handler removes the date/time information from HTTP headers of the response. In practice, the required application-specific knowledge is minimal and limited to parsing protocol headers; the payload of the request or response (*e.g.*, the HTTP body) is treated opaquely by the sketcher.

Whitelisting and blacklisting add flexibility to the handling of requests, but may require additional application-specific knowledge. One use of blacklisting that does not require such knowledge is to dynamically blacklist requests that exhibit a high frequency of transitions (*e.g.*, dynamic content). This allows the sketcher to avoid issuing fast reads that are very likely to fail. (We do not currently implement this optimization.)

3.4 Performance

In our analysis and evaluation, the sketcher is able to accommodate all read requests in its history table without evicting any entries. If needed, a replacement policy such as LRU may be used, but this is unlikely: our current implementation can store up to 22 million unique entries using less than 1GB of memory.

The performance savings of a sketcher come from the ability to execute fast, load-balanced reads whose responses match the entries of the history table. Thus, Prophecy and D-Prophecy are most effective in read-mostly workloads. We can estimate the savings by looking at the cost, in terms of per-replica processing time, of executing a read in these systems. Let t be the probability that a state transition occurs in a given workload. Let C_R be the cost of a replicated read and C_r the cost of a fast read (excluding any sketcher processing in the case of D-Prophecy), and let C_{hist} be the cost of computing a sketch and performing a lookup/update in a history table. Below, we calculate the expected cost of a read in Prophecy and D-Prophecy when used with a BFT replica group that uses PBFT's read optimization. For comparison, we include the cost of the unmodified BFT group; here, t' is the probability that a PBFT optimized read fails.

$$\begin{aligned}
\text{Prophecy:}\quad & [C_r + 2C_{hist}] + [t(NC_R + C_{hist})] \\
\text{D-Prophecy:}\quad & [C_r + (N-1)C_{hist}] + [t(NC_R + NC_{hist})] \\
\text{BFT:}\quad & [NC_r] + [t'NC_R]
\end{aligned}$$

The addends on the left and right of each equation show the cost of a fast read and a replicated read, respectively. The equations do not include optimizations that benefit all systems equally, such as separating agreement from execution [67]. Prophecy performs two lookups in the history table during a fast read (one before and one after executing the read), and one update to the history table during a replicated read. D-Prophecy performs a history table lookup at all but one replica server during a fast read, and an update to the history table of each replica server during a replicated read. These equations show that Prophecy operates at maximum throughput when there are no transitions, because only one replica server processes each request, as compared to over $2/3$ of the replica servers in the BFT system (assuming, idealistically, that only a necessary quorum of replica servers execute the optimized read, and the remaining replicas ignore it). Since $C_{hist} \ll C_r$ for non-null workloads—the former involves an in-memory table lookup, the latter an actual read—this is a factor of over $(2/3)N$ improvement. D-Prophecy's savings are similar for the same reason. Although t' may be significantly less than t in practice—given that PBFT optimized reads may still succeed even when a state transition occurs—our evaluation in §7 reveals that the benefit of PBFT optimized reads over replicated reads is small for real workloads. Finally, while Prophecy's throughput advantage degrades as t increases, we demonstrate in §8 that t is indeed low for popular web services.

4 Consistency Properties

Despite their relatively simple designs, the consistency properties of Prophecy and D-Prophecy are only slightly weaker than those of the replica group. In this section, we formalize the notion of *delay-once consistency* introduced in §2. Delay-once consistency is a derived consistency model; here, we derive it from linearizability [32], the consistency model of most BFT protocols, and obtain *delay-once linearizability*. Then, we show how Prophecy implements delay-once linearizability.

4.1 Delay-once Linearizability

A history of requests and responses executed by a service is linearizable if it is equivalent to a sequential history [40] that respects the irreflexive partial order on requests imposed by their real-time execution [32]. Request X precedes request Y in this order, written $X \prec Y$, if the response of X is received before Y is sent. Suppose one client sends requests (R^a, W^b, R^c) to the service and another client sends requests (W^d, R^e, R^f, W^g), with partial order $\{R^a \prec R^e, W^g \prec R^c\}$. Then a valid linearized history could look like the following:

$$\langle R^a_0, W^d_1, W^b_2, R^e_2, R^f_2, W^g_3, R^c_3 \rangle.$$

The R's and W's represent read and write requests, and subscripts represent the service state reflected in the response to each request (following [28]). In contrast to this history, the following is a valid delay-once linearizable history, though it is not linearizable:

$$\langle R^a_0, W^d_1, W^b_2, R^e_0, R^f_2, W^g_3, R^c_2 \rangle.$$

Requests R^e and R^c have stale responses because they do not reflect the state update caused by sequentially precedent writes (note that the staleness of R^e's response is discernible to the issuing client, whereas the staleness of R^c's response is not). At a high level, a delay-once history looks like a linearized history with reads that reflect the state of prior reads, but not necessarily prior writes. The manner in which reads can be stale is not arbitrary, however. Specifically, a history H is *delay-once linearizable* if the subsequence of write requests in H, denoted by $H|_W$, satisfies linearizability, and if read requests satisfy the following property:

> **Delay-once property**. For each read request R_x in H, let R_y and W_z be the read and write request of maximal order in H such that $R_y \prec R_x$ and $W_z \prec R_x$. Then either $x = y$ or $x = z$.

Delay-once linearizability implies both monotonic read and monotonic write consistency, but not read-after-write consistency. If $\prec_H$ is the partial order of the history H, delay-once linearizability respects $\prec_{H|_W}$ but not $\prec_H$, due to the possible presence of stale reads.

The delay-once property ensures two things: first, reads never reflect state older than that of the latest read (they are only *delayed* to *one* stale state), and second, reads that are updated reflect the latest state immediately. Thus, a system that implements delay-once consistency is *responsive*. To verify if a read in a delay-once consistent history H is stale, one can check the following:

> **Staleness indicator**. Given a read request R_x in H, let W_y be the write request of maximal order in H such that $W_y \prec R_x$. R_x is stale if and only if $x < y$.

The staleness property explains why object-based systems like web services fare particularly well with delay-once consistency. In these systems, state updates to one object are isolated from other objects, so staleness can only occur between writes and reads to the same object.

The above derivation of delay-once consistency is based on linearizability, but derivations from other consistency models are possible. For example, a weaker condition called read-after-write consistency also yields meaningful delay-once semantics.

4.2 Prophecy's Consistency Semantics

We now show that Prophecy implements delay-once linearizability when used with a replica group that guarantees linearizability, such as a PBFT replica group. A similar (but simpler) argument shows that D-Prophecy achieves delay-once linearizability, omitted here due to space constraints.

Prophecy inherits the system and network model of the replica group. When used with a PBFT replica group, we assume an asynchronous network between the sketcher and the replica group that may fail to deliver messages, may delay them, duplicate them, or deliver them out-of-order. Replica clients issue requests to the replica group one at a time; requests are retransmitted until they are received. We do not make any assumptions about the organization of the service's state; for example, the service may be a monolithic replicated state machine [39, 58] or a collection of numerous, isolated objects [32]. The sketcher may process requests concurrently. We model this concurrency by allowing the sketcher to issue requests to multiple replica clients simultaneously; the order in which these requests return from replica clients is arbitrary. Updates to service state may not be discernible or visible to the sketcher—*i.e.*, there may exist an external write channel—as discussed in §3.3. We show that Prophecy achieves delay-once linearizability despite concurrent requests and external writers.

Our analysis of Prophecy's consistency requires a non-standard approach because it is the sketcher, not the replica servers, that enforces this consistency, and because fast reads are executed by individual replicas. In particular, we introduce the notion of an *accepted history*. Let H_i for $1 \le i \le N$ be the history of all write requests executed by replica server i and all fast read requests executed by i that were accepted by the sketcher. Let R_s be the history of all replicated read requests accepted by the sketcher. An accepted history A_i is the union of H_i and R_s, for each replica server i. The position in A_i of each replicated read in R_s is well defined because all reads are accepted at a single location (the sketcher) and all replicated requests are totally ordered by linearizability. We claim that the accepted history A_i is delay-once linearizable.

To see this, observe that replicated requests satisfy linearizability because they follow the protocol of the replica group. The sketcher ensures that replicated reads update the history table according to this order by using the sequence numbers returned by the *replicated* interface. Further, the sketcher only accepts a fast read if it reflects the state of the latest replicated read. Since A_i contains all replicated reads accepted by the sketcher (not just those accepted by i), and since accepted fast reads never reflect new state, it follows that all fast reads in A_i must satisfy the delay-once property. While A_i may not contain all write requests accepted by the replica group (*e.g.*, if i is missing an update), this only affects i's ability to participate in replicated reads, and does not violate delay-once linearizability. Thus, we conclude that A_i is delay-once linearizable.

Limiting staleness via load balancing. Stale responses are returned by faulty replica servers or correct replica servers that are out-of-date. We can easily verify if an accepted history contains stale responses by checking the staleness indicator defined in §4. To limit the number of stale responses, the *fast* interface dispatches fast reads from all clients uniformly at random over the replica servers.[2] Let g be the fraction of faulty or out-of-date replica servers currently in the replica group. If g is a constant, then g^k, the probability that k consecutive fast reads are sent to these servers, is exponentially decreasing. For BFT protocols, $g < 2/3$ assuming a worst-case scenario where the maximum number of correct nodes are out-of-date. For a replica group of size 4, the probability that $k > 6$ is less than 1.6%.

5 Scale and Complex Architectures

This section describes extensions to the basic Prophecy model in order to integrate fault tolerance into larger-scale and more complex environments.

Scaling through multiple sketchers. In the basic system model of Prophecy (Figure 3), the sketcher is a single bottleneck and point-of-failure. We address this limitation by using multiple sketchers to build a sketching core, as follows. First, we horizontally partition the global history table, based on $s(q)$'s, into non-overlapping regions, *e.g.*, using consistent hashing [33]. We assign each region to a distinct sketcher, which we refer to as *response sketchers*. The partitioning preserves delay-once semantics because only a single sketcher stores the entry for each $s(q)$. Second, we build a two-level sketching system as shown in Figure 4, where the first tier of *request sketchers* demultiplex client requests. That is, given a request q, any of a small number of request sketchers computes $s(q)$ and forwards q to the appropriate response sketcher. Using a one-hop distributed hash table (DHT) [27, 33] to manage the partitioning works well, given the network's small, highly-connected nature. The response sketchers (the members of this DHT) issue requests to the replica group(s) and sketch the responses, ultimately returning them to the clients. (Importantly, the replica servers in Figure 4 need not be part of a single replica group, but may instead be organized into multiple groups.) The larger number of response sketchers reflects the asymmetric bandwidth requirements of network protocols like HTTP. We evaluate the scaling benefits of multiple response sketchers in §7.7.

Handling sketcher failures. The sketching core handles failure and recovery of sketchers seamlessly, because it can rely on the join and leave protocol of the underlying DHT. Since request sketchers direct client requests, they maintain the partitioning of the DHT. To preserve delay-once semantics, this partitioning must be kept consistent [10, 66] to avoid sending requests from the same region of the history table to multiple response sketchers. Prophecy's support for blacklisting simplifies this task, however. In particular, whenever a region of the history table is being relinquished or acquired between response sketchers, we can allow more than one response sketcher to serve requests from the same region provided the entire region is blacklisted (forcing all requests to be replicated). Once the partitioning has stabilized, the new owner of the region can unset the blacklist bit. As a result, membership dynamics can be handled smoothly and simply, at the cost of transient inefficiency but not inconsistency.

Mediating loosely-coupled groups. A sketching core can be shared by the multiple, loosely-coupled components that typically comprise a real service. Alternatively, components that operate in parallel can use Prophecy via dedicated sketchers. Components that operate in series, such as multi-tier web services, can use Prophecy prior to each tier. However, applying agreement protocols in series introduces nontrivial consistency issues. We leave this problem to future work.

6 Implementation

Our implementation of Prophecy and D-Prophecy is based on PBFT [12]. We used the PBFT codebase given its stable and complete implementation, as well as newer results [6] showing its competitiveness with Zyzzyva and

[2] We assume for simplicity that the random selection is secure, though in practice faulty replica servers may hamper this process. The latter is an interesting problem, but outside the scope of this paper.

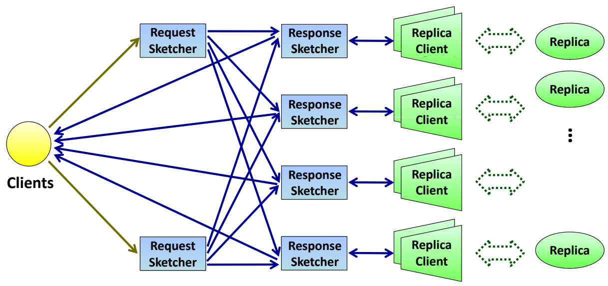

Figure 4: **Scaling out Prophecy using multiple sketchers.**

System	median	1st	99th
pr-PBFT	433	379	706
pr-PBFT-ro	296	255	544
Prophecy	256	216	286
Prophecy-100	617	553	768
PBFT	286	272	309
PBFT-ro	144	135	168
D-Prophecy	144	129	197
D-Prophecy-100	429	412	574

Table 2: **Latency in μs for serial null reads.**

other recent protocols (much more so than was originally indicated [38]). We implemented and compared three proxied systems (Prophecy, proxied PBFT without optimized reads, and proxied PBFT with optimized reads), as well as three non-proxied ("direct") systems (D-Prophecy, PBFT without optimized reads, and PBFT with optimized reads). In our evaluation, we will compare proxied systems only with other proxied systems, and similarly for direct systems, as the architectures and assumptions of the two models are fundamentally different. The proxied systems do not authenticate communication between clients and the sketcher, though they easily can be modified to do so with equivalent overheads.

We implemented a user-space Prophecy sketcher in about 2,000 lines of C++ code using the Tamer asynchronous I/O library [36]. The sketcher forks a process for each core in the machine (8 in our test cluster), and the processes share a single history table via shared memory. The sketcher interacts with PBFT replica clients through the PBFT library. The pool of replica clients available to handle requests is managed as a queue. The sketching function uses a SHA-1 hash [48] over parts of the HTTP header (for requests) and the entire response body (for responses). The proxied PBFT variants share the same code base as the sketcher, but do not perform sketching, issue fast reads, or create or use the history table.

We modified the PBFT library in three ways: to add support for fast reads (about 20 lines of code), to return the sequence numbers (about 20 LOC), and to add support for D-Prophecy (about 100 LOC). Additional modifications enabled the same process to use multiple PBFT clients concurrently (500 LOC), and modified the simple server distributed with PBFT to simulate a webserver and allow "null" writes (500 LOC), as null operations actually have 8-byte payloads in PBFT. We also wrote a PBFT client in about 1000 lines of C++/Tamer that can be used as a client in direct systems and as a replica client in proxied systems.

7 Evaluation

This section quantifies the performance benefits and costs of Prophecy and D-Prophecy, by characterizing their latency and throughput relative to PBFT under various workloads. We explore how the system's throughput characteristics change when we modify a few key variables: the processing time of the request, the size of the response, and the client's session length. Finally, we examine how Prophecy scales with the replica group size.

7.1 Experimental Setup

All of our experiments were run in a 25-machine cluster. Each machine has eight 2.3GHz cores and 8GB of memory, and all are connected to a 1Gbps switch.

The proxied systems are labeled Prophecy, pr-PBFT (proxied PBFT), and pr-PBFT-ro (proxied PBFT with the read optimization). The direct systems are labeled D-Prophecy, PBFT, and PBFT-ro (PBFT with the read optimization). Multicast and batching are not used in our experiments, as they do not impact performance when using read optimizations; all other PBFT optimizations are employed. Unless otherwise specified, all experiments used four replica servers, a single sketcher/proxy machine for the proxied systems, and a single client machine. The proxied experiments used 40 replica clients across eight processes at the sketcher/proxy, and had 100 clients establish persistent HTTP connections with the sketcher/proxy. The direct experiments used 40 clients across eight processes. These numbers were sufficient to fully saturate each system without degrading performance. All experiments use infinite-length sessions between communicating entities (except for the one evaluating the effect of session length). Throughput experiments were run for 30-second intervals and throughput was averaged over each second.

In some experiments, we report numbers for Prophecy-X or D-Prophecy-X, which signifies that the systems experienced state transitions $X\%$ of the time.

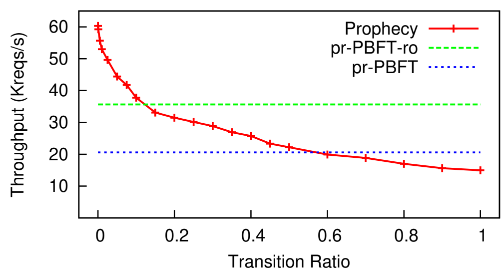

Figure 5: **Throughput of null reads for proxied systems (Prophecy, pr-PBFT, and pr-PBFT-ro).**

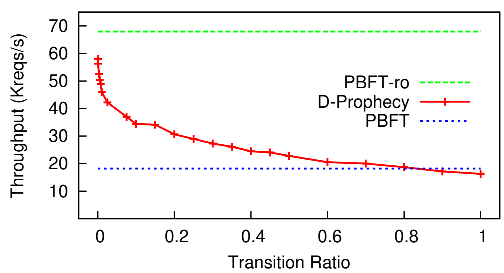

Figure 6: **Throughput of null reads for direct systems (D-Prophecy, PBFT, and PBFT-ro).**

7.2 Null Workload

Latency. Table 2 shows the median and 99th percentile latencies for 100,000 serial null requests sent by a single client. All systems displayed low latencies under 1*ms*, although the proxied systems have higher latencies as each request must traverse an extra hop. Prophecy, pr-PBFT-ro, D-Prophecy, and PBFT-ro all avoid the agreement phase during request processing and thus have notably lower latency than their counterparts. Prophecy-100 and D-Prophecy-100 represent a worst-case scenario where every fast read fails and is reissued as a replicated read.

Throughput. Figure 5 shows the aggregate throughput of the proxied systems for executing null requests. We achieve the desired transition ratio by failing that fraction of fast reads at the sketcher.

Since replica servers can execute null requests cheaply, the sketcher/proxy becomes the system bottleneck in these experiments. Nevertheless, Prophecy achieves 69% higher throughput than pr-PBFT-ro due to its load-balanced fast reads, which require fewer packets to be processed by replica servers. As the transition ratio increases, however, Prophecy's advantage decreases because fewer fast reads match the history table. For example, when transitions occur 15% of the time—a representative ratio from our measurement study in §8—Prophecy's throughput is 7% lower than pr-PBFT-ro's.

Figure 6 depicts the aggregate throughput of the direct systems. In this experiment, 40 clients across two machines concurrently execute null requests. D-Prophecy's throughput is 15% lower than PBFT-ro's when there are no transitions, and 50% lower when there are 15% transitions. D-Prophecy derives no performance advantage from its fast reads because the optimized reads of PBFT take no processing time, while D-Prophecy pays the overhead for sketching and history table operations.

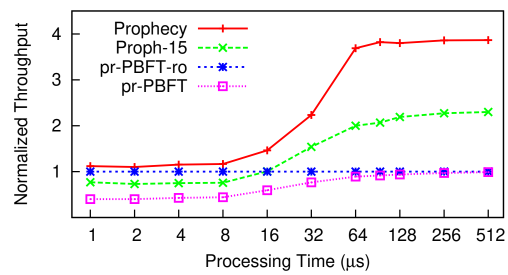

Figure 7: **Throughput of proxied systems as processing time increases, normalized against pr-PBFT-ro.**

7.3 Server Processing Time

The previous subsection shows that when requests take almost no time to process, Prophecy improves throughput only by decreasing the number of packets at each replica server, while D-Prophecy fails to achieve better throughput. However, when the replicas perform real work, such as the computation or disk I/O associated with serving a webpage, Prophecy's improvement is more dramatic.

Figures 7 and 8 demonstrate how varying processing time affects the throughput of proxied systems (normalized against pr-PBFT-ro) and direct systems (normalized against PBFT-ro), respectively. As the processing time increases—implemented using a busy-wait loop—the cost of executing requests begins to dominate the cost of agreeing on their order. This decreases the effectiveness of PBFT's read optimization, as evidenced by the increase in pr-PBFT's throughput relative to pr-PBFT-ro, and similarly between PBFT and PBFT-ro. At the same time, the higher execution costs dramatically increase the effectiveness of load balancing in Prophecy and D-Prophecy. Their throughput approaches 3.9 times the baseline, which is only 2.5% less than the theoretical maximum.

The effectiveness of load-balancing is more pronounced in Prophecy than in D-Prophecy for two main reasons. First, Prophecy's fast reads involve only one replica server, while D-Prophecy's fast reads involve all replicas, even though only a single replica actually executes the request. Second, Prophecy performs sketching and history table operations at the sketcher, whereas D-Prophecy implements such functionality on the replica servers, stealing cycles from normal processing.

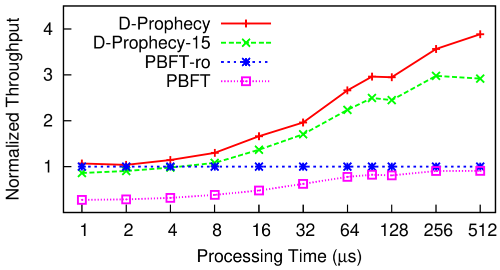

Figure 8: **Throughput of direct systems as processing time increases, normalized against PBFT-ro.**

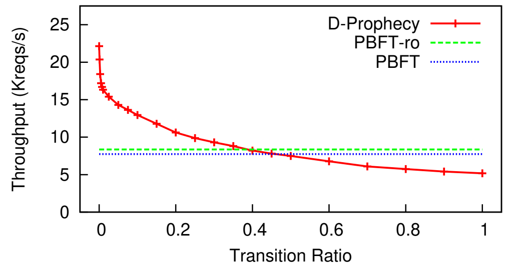

Figure 10: **Throughput of concurrent reads of a 1-byte webpage to Apache webservers for direct systems.**

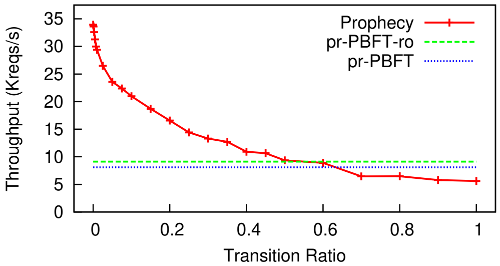

Figure 9: **Throughput of reads of a 1-byte webpage to Apache webservers for proxied systems.**

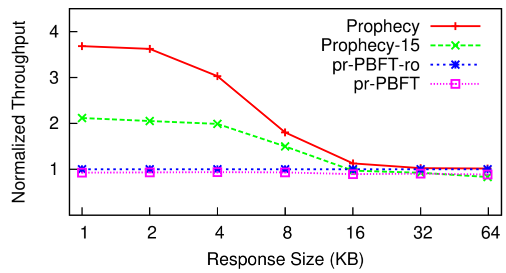

Figure 11: **Throughput of proxied systems as response size increases, normalized against pr-PBFT-ro.**

7.4 Integration with Apache Webserver

We applied Prophecy to a replica group in which each server runs the Apache webserver [7], appropriately modified to return deterministic results. Upon receiving a request, a PBFT server dispatches the request body to Apache via a persistent TCP connection over localhost.

Figure 9 shows the aggregate throughput of the proxied systems for serving a 1-byte webpage. When there are no transitions, Prophecy's throughput is 372% that of pr-PBFT-ro. At the representative ratio of 15%, Prophecy's throughput is 205% that of pr-PBFT-ro. The processing time of Apache is enough to dominate all other factors, causing Prophecy's use of fast reads to significantly boost its throughput.

Figure 10 shows the throughput of direct systems. With no transitions, D-Prophecy's throughput is 265% that of PBFT-ro, and 141% when there are 15% transitions.

In these experiments, the local HTTP requests to Apache took an average of $94\mu s$. For the remainder of this section, we use a simulated processing time of $94\mu s$ within replica servers when answering requests.

7.5 Response Size

Next, we evaluate the proxied systems' performance when serving webpages of increasing size, as shown by Figure 11. As the response size increases, fewer replica clients were needed to maximize throughput. At the same time, Prophecy's throughput advantage decreases as the response size increases, as the sketcher/proxy becomes the bottleneck in each scenario. Increasing the replica servers' processing time shifts this drop in Prophecy's throughput to the right, as it increases the range of response sizes for which processing time is the dominating cost. Note that we only evaluate the systems up to 64KB responses, because PBFT communicates via UDP, which has a maximum packet size of 64KB.

7.6 Session Length

Our experiments with direct systems so far did not account for the cost of establishing authenticated sessions between clients and replica servers. To establish a new session, the client must generate a symmetric key that it encrypts with each replica server's public key, and each replica server must perform a public-key decryption. Given the cost of such operations, the performance of short-lived sessions can be dominated by the overhead of session establishment, as we discussed in §2.2.

Figure 12 demonstrates the effect of varying session length on the direct systems, in which each request per session returns a 1-byte webpage. We find that the throughput of PBFT and PBFT-ro are indistinguishable for short sessions, but as session length increases, the cost of session establishment is amortized over a larger number of requests, and PBFT-ro gains a slight throughput advantage. Similarly, D-Prophecy achieves its full throughput advantage only when sessions are very long.

We do not evaluate the effect of session lengths in the proxied systems, because they currently do not authenti-

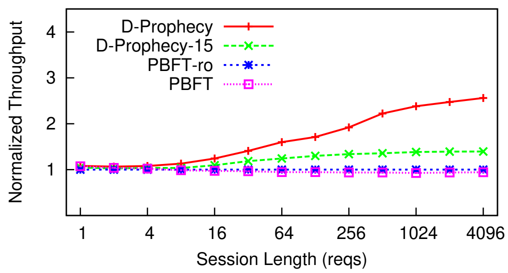

Figure 12: **Throughput of direct systems as session length increases, normalized against PBFT-ro.**

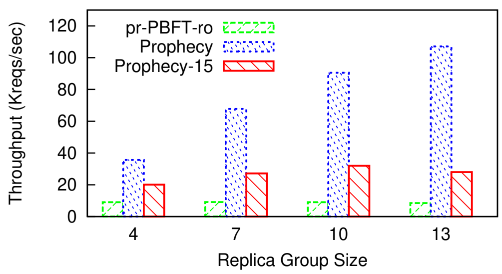

Figure 13: **Throughput of Prophecy and pr-PBFT-ro with varying replica group sizes.**

cate communication with the clients. Authentication can easily be incorporated into these systems, however, at a similar cost to Prophecy and pr-PBFT. That said, proxied systems can better scale up the maximum rate of session establishment than direct systems, as we observed in §3.3: each additional proxy provides a linear rate increase, while direct systems require an entire new replica group for a similar linear increase.

7.7 Scaling Out

Finally, we characterize the scaling behavior of Prophecy and proxied PBFT systems. By increasing the size of their replica groups, PBFT systems gain resilience to a greater number of Byzantine faults (*e.g.*, from one fault per 4 replicas, to four faults per 13 replicas). However, their throughput does not increase, as each replica server must still execute every request. On the other hand, Prophecy's throughput can benefit from larger groups, as it can load balance fast reads over more replica servers. As the sketcher can become a bottleneck in the system at higher read rates, we used two sketchers for a 7-replica group and three sketchers for a 10- and 13-replica group.

Figure 13 shows the throughput of proxied systems for increasing group sizes. Prophecy's throughput is 395%, 739%, 1000%, and 1264% that of pr-PBFT-ro, for group sizes of 4, 7, 10, and 13 replicas, respectively. Prophecy does not achieve such a significant throughput improvement when experiencing transitions, however. We see that a 15% transition ratio prevents Prophecy from han-

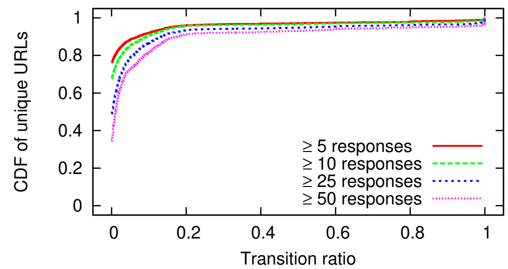

Figure 14: **A CDF of requests over transition ratios.**

dling more than $32,000$ req/s, which it achieves with a replica group of size 10. Thus, under moderate transition rates, further increasing the replica group size will only increase fault tolerance, not throughput.

8 Measurement Study of Alexa Sites

The performance savings of Prophecy are most pronounced in read-mostly workloads, such as those involving DNS: of the 40K names queried by the ConfiDNS system [52], 95.6% of them returned the same set of IP addresses every time over the course of one day. In web services, it is less clear that transitions are rare, given the pervasiveness of so-called "dynamic content".

To investigate this dynamism, we collected data from the Alexa top 25 websites by scripting a Firefox browser to reload the main page of each site every 20 seconds for 24 hours on Dec. 29, 2008. Among the top sites were `www.youtube.com`, `www.facebook.com`, `www.skyrock.com`, `www.yahoo.co.jp`, and `www.ebay.com`.[3] The browser loads and executes all embedded objects and scripts, including embedded links, JavaScript, and Flash, with caching disabled. We captured all network traffic using the tcpflow utility [19], and then ran our HTTP parser and SHA-1-based sketching algorithm to build a compact history of requests and responses, similar to the real sketcher.

Our measurement results show that transitions are rare in most of the downloaded data. We demonstrate a clear divide between very static and very dynamic data, and use Rabin fingerprinting [55] to characterize the dynamic data. Finally, we isolate the results of individual geographic "sites" using a CIDR prefix database.

8.1 Frequency of Transitions

For each unique URL requested during the experiment, we measured the ratio of state transitions over repeated

[3] While one might argue that BFT agreement is overkill for many of the sites in our study, our examples in the introduction show that Heisenbugs and one-off misconfigurations can lead to embarrassing, high-profile events. Prophecy protects against these mishaps without the performance penalty normally associated with BFT agreement.

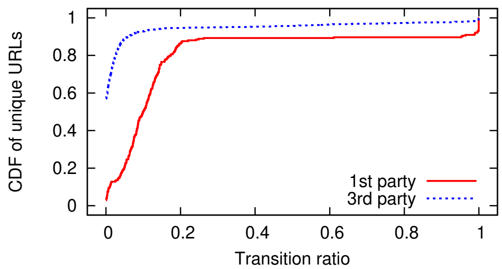

Figure 15: **A CDF over transition ratios of first-party vs. third-party URLs.**

requests. Figure 14 shows a CDF of unique URLs at different transition ratios. We separately plotted those URLs based on the number of requests sent to each one, given that embedded links generate a variable number of requests to some sites. (Where not specified, the minimum number of requests used is 25.) We see that roughly 50% of all data accessed is purely static, and about 90% of all requests have fewer than 15% state transitions. These numbers confirmed our belief that most dynamic websites are actually dynamic compositions of very static content. The same graph scaled by the average response size of each request yields very similar curves (omitted), suggesting that Figure 14 also reflects the total response throughput at each transition ratio.

Figure 15 is the same plot as Figure 14 but divided into first-party URLs, or those targeted at an Alexa top website, and third-party URLs, or those targeted at other sites (given that first-party sites can embed links to other domains for image hosting, analytics, advertising, etc.). The graph shows that third-party content is much more static than first-party content, and thus third-party content providers like CDNs and advertisers could benefit substantially from Prophecy.

The results in this section are conservative for two reasons. First, they reflect a workload of only three requests per minute per site, when in reality there may be tens or hundreds of thousands of requests per minute. Second, many URLs—though not enough to cause space problems in a real history table—saw only a few requests, but returned identical responses, suggesting that our HTTP parser was conservative in parsing them as unique URLs. An important characteristic of all of the graphs in this section is the relatively flat line across the middle: this suggests that most data is either very static or very dynamic.

8.2 Characterizing Dynamic Data

Dynamic data degrades the performance of Prophecy because it causes failed fast reads to be resent as replicated reads. Often, however, the amount of dynamism is small and may even be avoidable. To investigate this, we characterized the dynamism in our data by using Rabin fingerprinting to efficiently compare responses on either side of a transition. We divided each response into chunks of size 1K in expectation [47], or a minimum of 20 chunks for small requests.

Our measurements indicate that 50% of all transitions differ in at least 30% of their chunks, and about 13% differ in all of their chunks. Interestingly, the *edit distance* of these transitions was much smaller: we determined that 43% of all transitions differ by a single contiguous insertion, deletion, or replacement of chunks, while preserving at least half or no more than doubling the number of original chunks. By studying transitions with low edit distance, we can identify sources of dynamism that may be refactorable. For example, a preliminary analysis of around 4,000 of these transitions (selected randomly) revealed that over half of them were caused by load-balancing directives (*e.g.*, a number appended to an image server name) and random identifiers (*e.g.*, client IDs) placed in embedded links or parameters to JavaScript functions. In fact, most of the top-level pages we downloaded, including seemingly static pages like `www.google.com`, were highly dynamic for this exact reason. A more in-depth analysis is slated for future work.

8.3 Site-Based Analysis

A "site" represents a physical datacenter or cluster of machines in the same geographic location. A single site may host large services or multiple services. Having demonstrated Prophecy's ability to scale out in such environments, we now study the potential benefit of deploying Prophecy at the sites in our collected data. To organize our data into geographic sites, we used forward and reverse DNS lookups on each requested URL and matched the resulting IP addresses against a CIDR prefix database. (This database, derived from data supplied by Quova [54], included over 2 million distinct prefixes, and is thus significantly finer-grained than those provided by RouteViews [57].) Requests that mapped to the same CIDR prefix were considered to be part of the same site. Figure 16 shows an overlay of the transition plots of each site. From the figure, a few sites serve very static data or very dynamic data only, but most sites serve a mix of very static and very dynamic data. All but one site (`view.atdmt.com`) show a clear divide between very static and very dynamic data.

9 Related Work

A large body of work has focused on providing strong consistency and availability in distributed systems. In the fail-stop model, state machine replication typically used primary copies and view change algorithms to improve

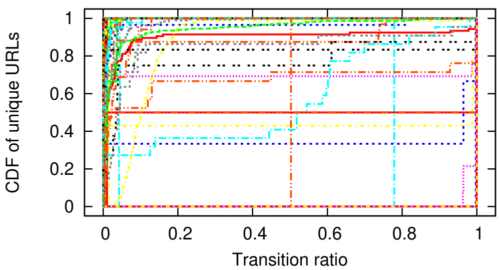

Figure 16: **A CDF of URLs over transition ratios for all sites for which CIDR data was available.**

performance and recover from failures [41, 50]. Quorum systems focused on tradeoffs between overlapping read and write sets [26, 31]. These protocols have been extended to malicious settings, both for Byzantine fault-tolerant replicated state machines [12, 42, 56], Byzantine quorum systems [1, 46], or some hybrid of both [17]. Modern approaches have optimized performance via various techniques, including by separating agreement from execution [67], using optimistic server-side speculation on correct operation [38], reducing replication costs by optimizing failure-free operation [65], and allowing concurrent execution of independent operations [37]. Prophecy's history table is motivated by the same assumption as this last approach—namely, that many operations/objects are independent and hence often remain static over time.

Given the perceived cost of achieving strong consistency and a particular desire to provide "always-on" write availability, even in the face of partitions, a number of systems opted for cheaper techniques. Several BFT replicated state machine protocols were designed with weaker consistency semantics, such as BFT2F [44], which weakens linearizability to fork* consistency, and Zeno [59], which weakens linearizability to eventual consistency. Several filesystems were designed in a similar vein, such as SUNDR [45] and systems designed for disconnected [29, 35] or partially-connected operation [51]. BASE [53] explored eventual consistency with high scalability and partition tolerance; the foil to database ACID properties. More recently, highly-scalable storage systems being built out within datacenters have also opted for cheaper consistency techniques, including the Google File System [25], Yahoo!'s PNUTS [16], Amazon's Dynamo [18], Facebook's Cassandra [20], eBay's storage techniques [61], or the popular approach of using Memcached [23] with a backend relational database. These systems take this approach partly because they view stronger consistency properties as infeasible given their performance (throughput) costs; Prophecy argues that this tradeoff is not necessary for read-mostly workloads.

Recently, several works have explored the use of trusted primitives to cope with Byzantine behavior. A2M [13] prevents faulty nodes from lying inconsistently by using a trusted append-only memory primitive, and TrInc [43] uses a trusted hardware primitive to achieve the same goal. Chun *et al.* [14] introduced a lightweight BFT protocol for multi-core single-machine environments that runs a trusted coordinator on one core, similar in philosophy to Prophecy's approach of extending the trusted computing base to include the sketcher.

Prophecy is unique in its application to customer-facing Internet services and its ability to load-balance read requests across a replica group while retaining good consistency semantics. Perhaps closest to Prophecy's semantics is the PNUTS system [16], which supports a load-balanced read primitive that satisfies timeline consistency (all copies of a record share a common timeline and only move forward on that timeline). Delay-once linearizability is strictly stronger than timeline consistency, however, because it does not allow a client to see a copy of a record that is more stale than a copy the client has already seen (whereas timeline consistency does).

There has been some work on using history as a consistency or security metric for particular applications. Aiyer et al. [4, 5] develop k-quorum systems that bound the staleness of a read request to one of the last k written values. Using Prophecy with a k-quorum system may be synergistic: Prophecy's load-balanced reads are less costly than quorum reads, and k-quorum systems can protect against an adversarial scheduler that attempts to hamper Prophecy's load balancing. The Farsite file system [3] uses historical sketches to validate read requests, but requires a lease-based invalidation protocol to keep sketches strongly consistent. The system modifies clients extensively and requires knowledge of causal dependencies (if these constraints are ignored, then D-Prophecy can easily be modified to achieve the same consistency as Farsite). Pretty Good BGP [34] whitelists BGP advertisements whose new route to a prefix includes its previous originating AS, while other routes require manual inspection. ConfiDNS [52] uses both agreement and history to make DNS resolution more robust. It requires results to be static for a number of days and agreed upon by some number of recursive DNS resolvers. Perspectives [63] combines history and agreement in a similar way to verify the self-signed certificates of SSH or SSL hosts on first contact. Prophecy can be viewed as a framework that leverages history and agreement in a general manner.

10 Conclusions

Prophecy leverages history to improve the throughput of Internet services by expanding the trusted middlebox between clients and a service replica group, while providing a consistency model that is very promising for many applications. D-Prophecy achieves the same benefits for more traditional fault-tolerant services. Our prototype implementations of Prophecy and D-Prophecy easily integrate with PBFT replica groups and are demonstrably useful in scale-out topologies. Performance results show that Prophecy achieves 372% of the throughput of even the read optimized PBFT system, and scales linearly as the number of sketchers increases. Our evaluation demonstrates the need to consider a variety of workloads, not just null workloads as typically done in the literature. Finally, our measurement study of the Internet's most popular websites demonstrates that a read-mostly workload is applicable to web service scenarios.

Acknowledgments

We thank our shepherd Petros Maniatis for helpful comments on earlier versions of this paper. Siddhartha Sen was supported through a Google Fellowship in Fault Tolerant Computing. Equipment and other funding was provided through the Office of Naval Research's Young Investigator program. None of this work reflects the opinions or positions of these organizations.

References

[1] M. Abd-El-Malek, G. Ganger, G. Goodson, M. Reiter, and J. Wylie. Fault-scalable byzantine fault-tolerant services. In *SOSP*, Oct. 2005.

[2] B. Adida. Helios: Web-based open-audit voting. In *USENIX Security*, July 2008.

[3] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI*, Dec 2002.

[4] A. S. Aiyer, L. Alvisi, and R. A. Bazzi. On the availability of non-strict quorum systems. In *DISC*, Sept. 2005.

[5] A. S. Aiyer, L. Alvisi, and R. A. Bazzi. Byzantine and multi-writer K-quorums. In *DISC*, Sept. 2006.

[6] L. Alvisi, A. Clement, M. Dahlin, M. Marchetti, and E. Wong. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *NSDI*, Apr. 2009.

[7] Apache HTTP Server. `http://httpd.apache.org/`, 2009.

[8] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *IEEE COMPSAC*, Nov. 1977.

[9] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4), 1985.

[10] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, Nov. 2006.

[11] M. Castro. *Practical Byzantine Fault-Tolerance*. PhD thesis, Mass. Inst. of Tech., 2000.

[12] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *OSDI*, Feb. 1999.

[13] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *SOSP*, Oct. 2007.

[14] B.-G. Chun, P. Maniatis, and S. Shenker. Diverse replication for single-machine Byzantine-Fault Tolerance. In *USENIX Annual*, June 2008.

[15] A. Clement, M. Marchetti, E. Wong, L. Alvisi, and M. Dahlin. BFT: The time is now. In *LADIS*, Sept. 2008.

[16] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB*, Aug. 2008.

[17] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *OSDI*, Nov. 2006.

[18] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lak-shman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, 2007.

[19] J. Elson. tcpflow—A TCP Flow Recorder. `http://www.circlemud.org/~jelson/software/tcpflow/`, 2009.

[20] Facebook. Facebook release cassandra: A structured storage system on a p2p network. `http://code.google.com/p/the-cassandra-project/`, 2008.

[21] Facebook. Scaling out. `http://www.facebook.com/note.php?note_id=23844338919`, Aug. 2008.

[22] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. *Theoretical Computer Science*, 220(1), 1999.

[23] B. Fitzpatrick. Memcached: a distributed memory object caching system. `http://www.danga.com/memcached/`, 2009.

[24] Flickr. Flickr phantom photos. `http://flickr.com/help/forum/33657/`, Feb. 2007.

[25] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *SOSP*, Oct. 2003.

[26] D. K. Gifford. Weighted voting for replicated data. In *SOSP*, Dec. 1979.

[27] A. Gupta, B. Liskov, and R. Rodrigues. Efficient routing for peer-to-peer overlays. In *NSDI*, Mar. 2004.

[28] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in five easy pieces. Technical Report 5, DEC Systems Research Centre, 1985.

[29] J. Heidemann and G. Popek. File system development with stackable layers. *ACM Trans. Comp. Sys.*, 12(1), Feb. 1994.

[30] J. Hendricks, G. Ganger, and M. Reiter. Low-overhead Byzantine fault-tolerant storage. In *SOSP*, Oct. 2007.

[31] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Trans. Comp. Sys.*, 4(1), Feb. 1986.

[32] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Sys.*, 12(3), 1990.

[33] D. Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *STOC*, May 1997.

[34] J. Karlin, S. Forrest, and J. Rexford. Pretty Good BGP: Improving BGP by cautiously adopting routes. In *ICNP*, Nov. 2006.

[35] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. Comp. Sys.*, 10 (3), Feb. 1992.

[36] E. Kohler. Tamer. `http://read.cs.ucla.edu/tamer/`, 2009.

[37] R. Kotla and M. Dahlin. High-throughput Byzantine fault tolerance. In *DSN*, June 2004.

[38] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *SOSP*, Oct. 2007.

[39] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Sys.*, 6(2), 1984.

[40] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9), Sept. 1979.

[41] L. Lamport. The part-time parliament. *ACM Trans. Comp. Sys.*, 16(2), 1998.

[42] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Sys.*, 4 (3), 1982.

[43] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda. Trinc: small trusted hardware for large distributed systems. In *NSDI*, Apr. 2009.

[44] J. Li and D. Mazières. Beyond one-third faulty replicas in Byzantine fault tolerant systems. In *NSDI*, Apr. 2007.

[45] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, Dec. 2004.

[46] D. Malkhi and M. Reiter. Byzantine quorum systems. In *STOC*, May 1997.

[47] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *SOSP*, Oct. 2001.

[48] NIS95. *FIPS Publication 180-1: Secure Hash Standard*. Natl. Institute of Standards and Technology, Apr. 1995.

[49] F. E. Notes. Needle in a haystack: efficient storage of billions of photos. `http://www.facebook.com/note.php?note_id=76191543919`.

[50] B. M. Oki and B. H. Liskov. Viewstamped replication: a general primary copy. In *PODC*, 1988.

[51] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible update propagation for weakly consistent replication. In *SOSP*, Oct. 1997.

[52] L. Poole and V. S. Pai. ConfiDNS: Leveraging scale and history to improve DNS security. In *WORLDS*, Nov. 2005.

[53] D. Pritchett. BASE: An ACID alternative. *ACM Queue*, 6(3), 2008.

[54] Quova. http://www.quova.com/, 2006.

[55] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Harvard Aiken Computation Laboratory, 1981.

[56] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *SOSP*, Oct. 2001.

[57] RouteViews. http://www.routeviews.org/, 2006.

[58] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computer Surveys*, 22(4), Dec. 1990.

[59] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: eventually consistent byzantine-fault tolerance. In *NSDI*, apr 2009.

[60] TechCrunch. Facebook source code leaked. `http://www.techcrunch.com/2007/08/11/facebook-source-code-leaked/`, Aug. 2007.

[61] F. Travostino and R. Shoup. eBay's scalability odyssey: Growing and evolving a large ecommerce site. In *LADIS*, Sept. 2008.

[62] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine faults in database systems using commit barrier scheduling. In *SOSP*, Oct. 2007.

[63] D. Wendlandt, D. G. Andersen, and A. Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In *USENIX Annual*, June 2008.

[64] B. Wester, J. Cowling, E. B. Nightingale, P. M. Chen, J. Flinn, and B. Liskov. Tolerating latency in replicated state machines through client speculation. In *NSDI*, Apr. 2009.

[65] T. Wood, R. Singh, A. Venkataramani, and P. Shenoy. ZZ: Cheap practical bft using virtualization. Technical Report TR14-08, University of Massachusetts, 2008.

[66] Yahoo! Hadoop Team. Zookeeper. `http://hadoop.apache.org/zookeeper/`, 2009.

[67] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *SOSP*, Oct. 2003.