Aegean: Replication beyond the client-server model
Extended technical report

Remzi Can Aksoy, Manos Kapritsos
University of Michigan
{remzican,manosk}@umich.edu

Abstract

This paper presents Aegean, a new approach that allows fault-tolerant replication to be implemented beyond the confines of the client-server model. In today's computing, where services are rarely standalone, traditional replication protocols such as Primary-Backup, Paxos, and PBFT are not directly applicable, as they were designed for the client-server model. When services interact, these protocols run into a number of problems, affecting both correctness and performance. In this paper, we rethink the design of replication protocols in the presence of interactions between services and introduce new techniques that accommodate such interactions safely and efficiently. Our evaluation shows that a prototype implementation of Aegean not only ensures correctness in the presence of service interactions, but can further improve throughput by an order of magnitude.

1 Introduction

This paper rethinks the design and implementation of replicated services in modern large-scale environments where services are no longer standalone, but are components of larger systems and frequently interact with other services.

Interactions between services are an integral part of today's computing [57]. Services, large and small, are typically used in conjunction with other services to build complex systems, like large-scale key-value stores [14, 24, 32], online shopping centers [4], data processing systems [23, 31], etc. In such environments, services frequently need to issue requests to other services. For example, when an online store processes a client checkout, it issues a *nested* request—i.e., a request that is issued while the original checkout request is being processed—to the client's credit card service, asking for a certain amount of money to be blocked. Similarly, a web server processing client HTTP requests frequently needs to issue nested requests to a backend database.

Infrastructure services follow a similar pattern: multiple components work together to provide a high-level service. Systems like HBase [32] and Google's Photon system for processing ad clicks [8] consist of multiple components, some of which need to interact with as many as three other components to process client requests. In the spirit of the *microservices* paradigm [7, 27, 34], Uber's infrastructure includes about 1000 interacting microservices [35, 53]. Sometimes multiple services make use of a shared service providing useful functionality, such as a Single-Sign-On (SSO) service.

These interacting services are often mission-critical and need to be replicated for high-availability; e.g., the 2PC coordinator in Spanner [19], the HBase Master node [32], the Distributed Sagas coordinator [54], online stores, banking applications, etc. If one tries to replicate these services, however, one faces an unpleasant realization: our current replication protocols are both *inefficient* and *incorrect* in the presence of service interactions.

Our replication protocols, it turns out, were designed for the client-server model, where clients issue requests to a standalone service. These protocols dutifully follow the recipe of State Machine Replication [56] and provide clients the abstraction that they are interacting with a *single correct machine.* In a world where services interact, however, it is no longer enough to provide this abstraction towards the clients alone. If correctness is to be maintained, the abstraction must be provided towards *any* external entity that can observe the replicated service's state; e.g., any services that the replicated service must issue a nested request to.

The implications of this observation are far-reaching. As far as we can tell, all existing replication protocols, while perfectly fine in the client-server model, can run into a number of problems if applied naively in an environment where services interact. This includes Primary-Backup [12, 22], the myriad variants of Paxos [42], PBFT [13], Zyzzyva [40], Eve [38], Rex [29], Crane [21], XFT [47] and even the simple approach of providing replication in time by having a server log its requests on an external fault-tolerant log [11, 37, 41]. In the next section we give concrete examples of how interactions violate the correctness of these replication protocols. One of the contributions of this paper is to rethink the design of replication protocols and to ensure correctness in the presence of service interactions.

But correctness is not the only concern here. Using traditional replication protocols in such a *multi-service* environment would still lead to an inefficient implementation. Consider, for example, how a Paxos-based replicated service *A* would execute a client request that issues a nested request while executing. Paxos, like most replication protocols, is bound by sequential execution. That is, in order to guarantee replica consistency, requests are executed in a well-defined order and each request must finish executing before the next one can start. In a multi-service setting, this means that service *A* must remain idle while its nested request is being processed at service *B*, resulting in an inefficient implementation. If, as is often the case with real services [57], the chain of services is longer than just two, the performance loss only grows worse. In this paper, we reconsider the need for sequential execution and propose a deterministic alternative that is better suited for a multi-service environment.

In the absence of a comprehensive solution to the problem of service interactions, existing systems take one of the following two approaches. The simplest is to ignore their implications and accept the resulting inconsistencies. Google, for example, uses this approach in some of their services and try to deal with the inconsistencies at the application level [1]. If one does not wish to give up consistency, the alternative is to design a custom protocol to regulate these interactions. For example, Salus [61] creates a custom protocol to allow its replicated region servers to interact. The resulting protocol is quite complicated and does not generalize to other settings. Similarly, in Farsite [3] groups of replicated nodes can issue requests to other such groups. To simplify the design, Farsite uses a custom protocol based solely on message passing and avoids using nested requests altogether. While this circumvents the problem, it is hardly practical, as most developers prefer the simplicity of RPCs to having to decompose their requests into a sequence of independent state machine transitions. This paper aims to establish a set of principles and techniques that will facilitate the replication of services in multi-service environments, without resorting to ad-hoc solutions.

Contributions This paper rethinks the design of replication protocols beyond the client-server model. It proposes new techniques that address the shortcomings of existing replication protocols and allow replicated services to interact without harming performance or correctness. In particular, we make the following contributions:

- **Problem statement** We identify the shortcomings of existing replication protocols in the presence of service interactions and pinpoint three challenges that a comprehensive solution must address (§2).
- **Ensuring correctness** We introduce three new techniques, *server-shim*, *response-durability* and *spec-tame*, which together aim to provide the *single correct machine* (hereafter, *SCM*) abstraction (§4).
- **Designing for performance** We introduce *request pipelining*, a novel technique that achieves replica consistency without resorting to sequential execution. Request pipelining thus allows replicated services to issue nested requests without having to remain idle waiting for the response (§5).
- **Evaluation** We built *Aegean*, a replication library that implements the above techniques. We evaluate its performance using the TPC-W benchmark and two interacting services. Our experiments indicate that, not only does Aegean ensure the correctness of replicated services in a multi-service setting, but it can also increase their throughput by an order of magnitude (§7).

2 The consequences of interactions

We now explain in more detail how service interactions can violate the correctness of existing replication protocols. We will illustrate these violations using a simple setting in which a number of clients send requests to a replicated online store—which we call the *middle* service. As part of executing some client requests (e.g., checkout requests), the online store must make a *nested request* to another, unreplicated service (e.g., a credit card service)—that we call the *backend* service.

Despite its simplicity, this setting is representative of the call pattern found in many real applications. This is, for example, how a travel metasearch engine—e.g. Expedia [25], Kayak [39]—would work. This is the pattern one would observe if they replicated the Master node of HBase [32] or the coordinator in Distributed Sagas [54] to increase availability; etc. The only difference is that in these real applications, the interactions are even more complicated than our simple setting: the middle service may interact with multiple backend services, the chain of nested requests may be longer, and a request may issue any number of nested requests.

Our basic "client→middle→backend" setting is deliberately simple, to allow for clear exposition and also to demonstrate the fundamental nature of the problems we discuss. Our solutions and prototype, however, are motivated by real-world applications and thus they are not limited to this simple setting: they support an arbitrary number of nested requests per client request (including no nested requests), longer chains of services, interactions with multiple backend services and interactions with replicated backend services. We do not currently support circular service dependencies, as they can introduce deadlocks.

Note that, if performance is not a concern, some of the correctness issues we discuss can have simple, if inefficient, solutions. In this paper we focus not only on ensuring correctness in the presence of service interactions, but also on implementing these interactions efficiently.

Primary-backup Let us first consider primary-backup [12, 22], a class of replication protocols where the primary executes client requests and forwards state updates to the backup. Once the backup acknowledges receiving an update, the primary sends the corresponding response to the client. Now consider the implications of having a middle service replicated with primary-backup. Remember that only the primary executes requests and thus only the primary sends nested requests to the backend service.

Consider the case where the primary sends a nested payment request and then crashes. This will cause an inconsistency: the backend service will receive the request and process it, but the state of the middle service does not reflect that. The backup will eventually time out and become the new primary, having no idea that such a request was ever made. What is worse, it has no way of replaying the primary's execution; it does not even know what requests the primary executed. The backup may therefore end up re-issuing that nested request, issuing a slightly different request, or even realizing that the item is sold out and returning an error message to the client—who has already been charged.

The fundamental problem here is that the primary sends a nested request, essentially performing an *output commit* without first ensuring that its state is replicated at the backup. This is an example of the principle we mentioned earlier: when services interact, it is imperative that the SCM abstraction be provided towards *any* external entity that observes the replicated service's state.

Paxos-like protocols Paxos [42, 43] is the most popular replication protocol, with a large number of variants [10, 26, 36, 44, 45, 48, 50, 52, 60]. The problems we highlight here apply to all variants of Paxos, including all Byzantine replication protocols (e.g., PBFT [13], UpRight [15], XFT [47]) and the multithreaded execution of Crane[1] [21]. These protocols use active replication, where replicas first agree on the order of requests and then execute them in that order.

In a multi-service setting, these protocols run into several problems. Using active replication means that all replicas execute requests and thus, if no precautions are taken, the backend will receive and execute multiple copies of each nested request. The good news is that these requests are now guaranteed to be identical, making it easy to detect duplicates.

Duplicate detection, however, is not enough. Consider one of the replicas executing a batch of 10 requests, each of which makes a nested request to the backend service. After executing all 10 requests, the replica sends its responses back to the client and then crashes. When other replicas try to execute these same 10 requests, they must reach the same state as the failed replica—as this state has already been exposed to the client.

But how can they? Even if the backend service detects that these are duplicate requests and refuses to re-execute them, this does not help the replicas of the middle service, which are now stuck, with no way to get the reply to their nested requests. Here, one may consider employing existing techniques such as using a *reply cache* at the backend [46]; i.e. storing the latest response from each client. Such techniques are only part of the solution, however, as they do not consider the implications of a replicated client. Indeed, in such a setting one client replica may make an unbounded number of requests and overwrite the cache before another replica makes its first request.

Needless to say, BFT versions of Paxos introduce additional problems. For example, it is clearly not safe to allow any single replica to issue nested requests to the backend service, for fear of that replica being malicious.

[1]Crane also suffers from divergence if multiple replicas receive responses to their nested requests in different orders.

Finally, correctness is not the only problem for Paxos-like protocols. As we discussed earlier, their reliance on sequential execution results in significant inefficiencies in a multi-service setting. In Section 5 we rethink the use of sequential execution in this setting and propose *request pipelining*, an alternative mode of execution that can increase throughput up to 9x compared to sequential execution.

Speculative execution In the pursuit of high performance, some replication protocols use various forms of speculation. For example, Zyzzyva [40] executes requests before the replicas have reached agreement on the ordering of requests. Eve [38] takes speculation even further, speculating not only on the order of requests, but also on the result of a non-deterministic execution of the requests in a multithreaded environment.

While speculation can be an important tool towards improving performance, its use in multi-service settings, if left unchecked, can threaten the correctness of the system. The premise behind speculation is that it is acceptable to execute requests speculatively, as long as the client—the only external observer in the client-server model—is not exposed to the inconsistent states that result from mis-speculation. This is why speculative systems like Eve and Zyzzyva employ "repair" mechanisms to roll back the application state, in case the speculation is not successful. In a multi-service setting, however, the client is not the only external observer. When a middle service A makes a nested request to a backend service B, then B is also implicitly exposed to the internal state of A. If A made the nested request as part of a speculative execution, but the speculation was not successful, A should be able to roll back its application state; but it cannot take back the nested request it sent to B, nor revert the cascading effects of that nested request on the state of B and on any responses that B may have sent to its other clients.

This is yet another example of the basic principle behind this paper: when services interact it is not enough to provide the SCM abstraction towards the client; it must be provided towards any external entity that can observe the replicated service's state.

2.1 Putting it all together

The bad news so far is that our existing ways of achieving replication do not fare well when the replicated service is not standalone. What is more, each replication protocol seems to fail in its own particular way. Fortunately, that is not entirely true. If we look carefully at the above examples, we can distill these seemingly disparate issues into three key challenges raised by service interactions. Unsurprisingly, all three challenges are related to re-establishing the SCM abstraction.

Replicated client In the client-server model, a client is a single machine that issues requests to a server. In the multi-service setting, however, where a replicated service may

issue requests to other services, it effectively functions as a *replicated client*. While multiple replicas of the client may send their own copies of a request, these should be logically treated as a single request to maintain the SCM abstraction.
Durability of nested responses We refer to the response to a nested request as a *nested response*. When a middle service replica receives a nested response, it cannot simply finish the execution of the corresponding request and send the response back to the client. If the replica does so and then crashes, the other replicas may not be able to identify what was the response to that particular nested request. Instead, the replica should first ensure that the nested response has been received by enough middle service replicas so as to be considered durable.
Speculative execution A nested request should *never* be made based on speculative state. Doing so risks exposing to the backend service an uncommitted state which may later be rolled back, violating the SCM abstraction.

Section 4 presents our solutions to these challenges. In particular, we introduce three novel techniques, *server-shim*, *response-durability* and *spec-tame*, which address each of these challenges, respectively.

2.2 Alternative designs

The above problems manifest when a replicated middle service issues nested requests to a backend service. It is important to note that there are ways to achieve fault tolerance in middle services without replicating the middle service itself, thus avoiding the problems we mentioned above. In particular, one may not need to replicate a service if the following conditions hold: a) the service is purely stateless, and b) one only cares about tolerating crash failures. If such a stateless service crashes, one can restore it by starting another instance of the service [14, 32].

Such stateless designs often rely on a fault-tolerant backend store [9, 19]—to keep all application state—and sometimes use a reliable message queue [6] or logging service [37, 41]—to ensure reliable message exchange between the middle and backend service. For example, Amazon Web Services (AWS) provides applications with both a Simple Queue Service [6] and a Simple Notification Service [5] to facilitate such interactions. If a stateless service wants to change the application state, it can then issue requests to a backend storage service and receive the results via one of these abstractions. One of the drawbacks of this approach is that all such state updates, no matter how fine-grain, must be explicitly encoded in the state of the backend service and each such update must be issued through an explicit message between the two services.

3 System model

The concepts and techniques presented in this paper are fully general: they apply to both synchronous and asynchronous systems, and can be configured to tolerate all types of failures, from crashes to Byzantine failures. We primarily target asynchronous systems, where the network can arbitrarily delay, reorder or drop messages without imperiling safety. For liveness, we assume the existence of synchronous intervals during which messages sent between correct nodes are received and processed with bounded delay.
Backend service We consider both replicated and non-replicated backend services. Of course, not replicating the backend service is inadvisable, since a single machine failure can render it unavailable. This may seem like a weird setting—a high-availability service relying on a low-availability one—but in practice this can happen if the backend service is an optional service (e.g. a side banner on a web page, a plugin, or an optimization/refinement service) and the middle service can simply timeout on that service and continue operating without it.
Failure model To accommodate all failure types, we adopt the hybrid failure model of UpRight [15]. The system is guaranteed to be *live*—i.e., to provide a response to client requests—despite a total of up to u failures of any type. Additionally, the system is *safe*—i.e., ensures that all responses that are accepted by clients are correct—despite up to r commission failures and any number of omission failures[2]. Omission failures occur when a node omits some of the steps in its execution, but other than that adheres to the protocol; e.g., it crashes and takes no more steps or it neglects to send a message. Commission failures occur when a node performs actions that do not adhere to the protocol; e.g., it behaves maliciously. The union of omission and commission failures are Byzantine failures. Finally, we assume that faulty nodes cannot subvert cryptographic primitives, such as inverting SHA-256.
Correctness Previous replication protocols which employ sequential execution [2, 13, 16, 20, 40, 42] typically provide linearizability of requests [33]. This has led to the misconception that linearizability is the consistency guarantee that should be provided by a replication protocol. While this may be true when sequential execution is assumed, it is *not* true in general. For example, protocols like Primary-Backup [12, 22], Rex [29] and Eve [38] which support multithreaded execution have already moved away from linearizability, for the simple reason that a parallel execution of requests is not necessarily equivalent to *any* sequential ordering of those requests.

In Aegean we aim at a more general definition of correctness, that is not bound to sequential execution. We define correctness as *indistinguishability* from a single correct server; namely that the outputs of a replicated service should

[2]To quickly convert to the traditional "f" notation, one can substitute $u = f$, $r = 0$ when tolerating benign failures, and $u = r = f$ when tolerating Byzantine failures.

be indistinguishable from the outputs provided by an unreplicated version of the service. Note that this definition does not prevent achieving a linearizable execution. In fact, if the unreplicated version is executing requests sequentially, then the replicated service must guarantee linearizability. At the same time, this definition allows for other modes of execution; e.g. multithreaded execution.

4 Ensuring correctness

In this section we introduce three techniques, *server-shim*, *response-durability* and *spec-tame*, which aim to address the shortcomings of existing replication protocols and allow services to be safely replicated in a multi-service setting.

4.1 Server shim

As we discussed in Section 2, one of the challenges—in fact, the simplest one to address—raised by replicating a middle service that may issue nested requests to a backend service is that the middle service effectively functions as a *replicated client* to the backend service. If the middle service uses active replication, the backend will receive multiple copies of each nested request and should avoid executing redundant copies. Moreover, the backend should ensure that all replicas of the middle service can access the results of previously executed nested requests, if they need to.

Dealing with the complexities of a replicated client requires modifications to the backend service. It becomes imperative, then, to implement this functionality as simply and generically as possible, to avoid having to re-implement it for every backend service.

We propose a simple abstraction, *server-shim*, that abstracts away from the backend service the complexities of dealing with a replicated client, while at the same time providing replicas of the middle service with all the information they need. This idea mirrors the shim layer used in clients of traditional replication systems to abstract away from the client the complexities of dealing with a replicated server. The server-shim code runs independently at each replica, without any need for coordination. It is essentially a "filter" that aims to abstract away from each replica the complexity of dealing with a replicated client. The server-shim deals with the following aspects of replicated clients:

Receiving requests The server-shim authenticates each incoming request separately in order to ascertain the identity of the sender. It does not, however, forward all requests to the replica for processing immediately. Instead, it waits until it receives a *quorum* of matching requests from a replicated client—much like the quorum of matching responses required by a client of a replicated service. The size of the quorum depends on the failure model. Once a quorum of matching requests has been collected, the shim will forward the request to the replica for processing and will ignore any redundant copies sent by the remaining client replicas.

Sending responses When the backend service generates a response to a nested request, the server-shim is responsible for broadcasting it to all replicas of the middle service which sent the corresponding nested request. Merely sending the request to all replicas, however, is not enough to ensure that they all receive it, as messages may be lost in the network. Similar to traditional replicated protocols, the shim uses a per-client-thread reply cache to keep track of the latest response and resends it if needed. Importantly, by ensuring the durability of nested responses at the middle service—discussed below—we ensure that the reply cache of the backend only needs to store the latest response for each execution thread at the middle service.

4.2 Durability of nested responses

In the client-server model, the only input to a replica's state-machine execution are the client requests. In a multi-service setting, however, there is an additional source of input: the responses to its nested requests—i.e., *nested responses*. Just as traditional replication protocols must ensure that all inputs are durably logged—e.g., in the form of the Paxos log—before performing an output commit to the client, so must replication protocols in the multi-service setting ensure that all inputs, including nested responses, are durably logged before performing an output commit to the client or to a backend service.

In Aegean, when a middle service replica receives a nested response, it will forward an acknowledgment to all other replicas. Only when receiving $u + 1$ such acknowledgments, including its own, will a replica consider the nested response durable and perform any corresponding output commits. Unfortunately, this explicit acknowledgment phase results in additional latency per request. In Section 6 we show how we can leverage the backend service to get the same durability guarantee while eliminating this additional latency.

4.2.1 Timeout actions

Some applications do not always wait to receive a nested response, but may instead timeout. Consider for example a replicated 2PC coordinator, like the one used in Spanner [19], which sends a request to all participants. If some participants have not responded within a certain timeout, the coordinator will decide to abort the transaction. In these cases, the problem of durability of nested responses is transformed into a problem of *agreement* on the nested responses. There is no longer a single potential outcome to each nested request, but two: either a nested response was accepted or the middle service timed out.

To ensure that all replicas agree on the outcome of the nested request, we upgrade our acknowledgement phase into an instance of agreement (consensus). Once again, we don't actually have to run an explicit agreement in the common case: we leverage the backend to perform this agreement without incurring any additional latency (Section 6). We only perform this agreement phase explicitly when the replicas time out on the backend.

4.3 Taming speculation

Speculation is an approach for achieving high performance in a replicated system. For example, Zyzzyva [40] speculatively executes requests before reaching agreement on their order, while Eve [38] speculates not only on the order of requests, but also on the result of a non-deterministic, multi-threaded execution. As we discussed in Section 2, sending nested requests based on speculative state is dangerous: if speculation fails, we can employ a rollback mechanism to revert the application state to a previous version; but we cannot "unsend" the nested request, which may have already caused visible changes to the state of the backend service.

Interestingly, this problem also applies to replication protocols that are not typically considered speculative, like primary-backup [12, 22] and Rex [29]. Take primary-backup, for example. The primary can execute requests in any order and even in parallel. This execution is actually speculative, since there is no prior agreement on which requests should be executed, and how. In the client-server model this is fine since the execution has no visible side-effects; if the primary fails before updating the backup, the backup will take over and execute its own requests. In the multi-service setting, however, the primary's execution *does* have side-effects—the nested requests—which cause inconsistency.

The fundamental problem here is that speculation violates the SCM abstraction, as it exposes speculative state to the backend service. The brute-force approach to solving this problem would be to make the backend service aware of the existence of speculation. This would force the backend service to implement a rollback mechanism, as well as an isolation mechanism for preventing its other clients from observing speculative state—essentially requiring full transactional support. We consider this an impractical solution, especially since, as we discussed in Section 1, services frequently have to interact with multiple other services. This brute-force approach would require all such backend services to add transactional support; a significant undertaking that would most likely be a show-stopper. Ideally we would like to be able to enjoy the performance benefits of speculation at the middle service, without having to affect the backend service.

To that end we introduce a new technique, *spec-tame*, that allows replicated services to employ speculative execution, while still providing the SCM abstraction to unmodified backend services. Spec-tame is based on the insight that nested requests are a type of output commit to an external observer. It is therefore possible to employ speculation within the service, as long as the speculation is *resolved* before the output commit happens. Essentially, spec-tame applies the idea of "Rethink the sync" [51] to interacting replicated services.

The principle of taming speculation—i.e., using it internally, but resolving it before performing output commit—is generally applicable to any replicated service that employs speculation as part of its execution. We expect, however, that the particular way in which this principle will be applied depends on the particular protocol used to replicate the middle service. In the remainder of this section, we will show how the spec-tame technique can be applied to tame the speculative execution of Eve [38]. We chose to demonstrate spec-tame on Eve because of its generality and the practical significance of multithreaded execution. We also wanted to demonstrate that spec-tame is applicable even to complex execution schemes, like that of Eve.

4.3.1 Case study: taming Eve's speculation

Eve [38] is a protocol for replicating multithreaded services. Eve uses the execute-verify architecture, where replicas speculatively execute a set of requests, called a *parallelBatch*, in parallel and then verify whether they reached the same state and produced the same responses. Eve uses a Merkle tree to compute a hash of the application state and its pending responses. At the end of each batch of requests each replica sends this hash to the verification stage to determine whether replicas have diverged. If the verification stage detects that the replicas have diverged, it will ask the replicas to roll back the application state and temporarily employ sequential execution to ensure progress.

Essentially, Eve's approach is to resolve the speculation at the end of a batch, which serves as a deterministic point in the execution at which replicas will check whether the speculation has succeeded. Of course, doing so in a multi-service environment would be unsafe: the nested requests sent during execution of the batch would be based on speculative—and thus potentially inconsistent—state.

In Aegean, we refine Eve's approach by applying the spec-tame technique. In other words, we need to identify a deterministic point in the execution that precedes the sending of all nested requests sent by the execution threads of Eve. To that end, we leverage the fact that the very point where the original request needs to send a nested request is—by the fundamental assumption of state machines being deterministic in SMR—a deterministic point in the execution.

At the high-level, then, multithreaded execution in Aegean works as follows. When presented with a parallelBatch, replicas will start executing requests in parallel, using one thread per request, just like in Eve. Each thread will keep executing until it reaches a natural stopping point, which we call a *barrier*. The barrier is either the sending of a nested request—more precisely, right before the nested request is sent—or the end of the parallelBatch. When a thread hits the barrier, it will wait until all other threads hit the barrier, as well. Essentially, the barrier represents a deterministic point across the execution of all threads, at which speculation can be resolved. Figure 1(a) illustrates the concept of the barrier

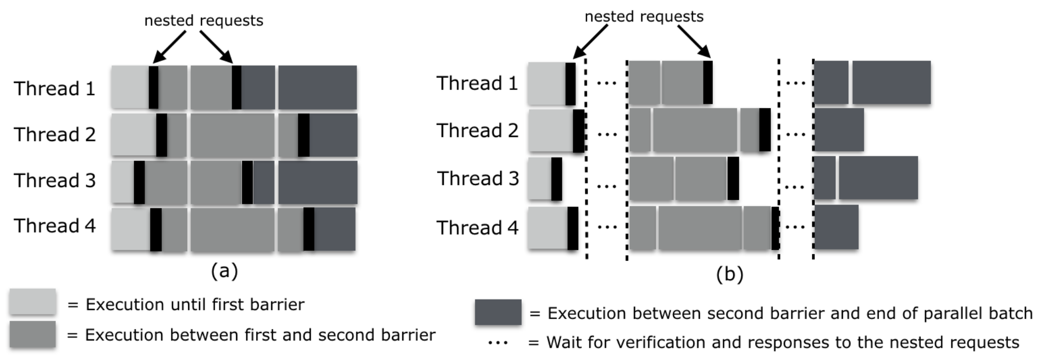

nested requests
Thread 1
Thread 2
Thread 3
Thread 4
(a)
nested requests
Thread 1
Thread 2
Thread 3
Thread 4
(b)
= Execution until first barrier
= Execution between first and second barrier
= Execution between second barrier and end of parallel batch
= Wait for verification and responses to the nested requests

Figure 1. The concept of a barrier and how threads use it to detect when speculation should be resolved. Figure (a) shows a parallelBatch of twelve requests being executed by four threads. Some requests need to make a nested request, while some do not. We use three different colors to indicate how the nested requests form two barriers that divide the execution into three parts. Figure (b) illustrates how the execution would proceed in real time, where threads must wait until all other threads hit the barrier, before proceeding to resolve the speculation and send their nested requests.

using a parallel execution with four threads and Figure 1(b) illustrates how threads wait for each other during execution.

When all threads hit the barrier, the replica will resolve the speculation similarly to Eve. It will compute a hash of the application state and responses using the Merkle tree and send the hash to the verification stage—which is identical to Eve's verification—to determine whether replicas have diverged. In addition to the state and responses, however, the hash will also include the nested requests that the replicas are about to send. Only if the verification stage determines that the replicas have converged—i.e., the Merkle hashes are identical—will the replicas mark their state as stable and send the nested requests. If the verification stage detects divergence, it will ask replicas to roll back to the latest stable state and retry execution from that point—sequentially this time, to ensure that replicas converge.

4.3.2 Subtleties of Aegean

The verification stage of Aegean is identical to that of Eve. In particular, if it can identify a state produced by a correct replica, it will ask replicas to commit their state. This will happen if it receives a quorum of $max(u, r) + 1$ matching Merkle hashes. Otherwise the verification stage will ask for a rollback. Note that, even when a quorum is collected, some replicas may have diverged, due to non-determinism in their execution. These replicas will simply ask for a state transfer from one of the replicas that committed, just like in Eve. There are some aspects of Aegean, however, that require special attention.

Fine-grain rollback In Eve, a rollback happens at the boundaries of a batch. If executing a batch led to divergence, each replica uses a multi-version, copy-on-write mechanism to revert the state to what it was at the beginning of the batch, and resumes execution of the requests, albeit in a sequential manner. In Aegean, however, rollback is more involved than that.

Consider, for example, the execution in Figure 1(b). If the replicas reach the first barrier without diverging, the verification will allow them to mark that state as stable and send the corresponding nested requests. If the replicas diverge before hitting the second barrier, then the verification will determine that a rollback is required. But what should the state be rolled back to? Since the replicas have already sent the nested requests at the first barrier, they cannot rollback the state any farther back than that—that was the whole point of taming speculation, after all. If they roll back the state to the first barrier, however, this means that each thread must resume the execution in the middle of the request. This means the thread must not only rollback the application state, but it must recreate the exact execution stack it was using when it hit the barrier.

Section 6 describes how Aegean implements rollback with this additional requirement. The important thing to note is that the application developer does not need to worry about the complexities of this new rollback mechanism; the replication library of Aegean takes care of the rollback automatically.

Late divergence Another subtlety that must be attended to is that of *late divergence*. As we said above, Eve resolves speculation by performing verification at the batch boundaries. In essence, it uses the verification stage to perform agreement on (a) the contents of the batch and (b) the way the batch was executed in a multithreaded manner. If agreement is not reached, the state is rolled back and a new batch is formed and executed sequentially. In Aegean, however, verification can also be performed in the middle of a batch. This creates the following complication.

Consider the case where the primary replica—the one responsible for forming batches—suffers from a commission failure, so that it sends different batches to all replicas. In particular, all requests in the batch are identical across replicas, except one request, which is different across all replicas.

If that "trap" request lies towards the end of the batch, the replicas may still succeed in converging all through the first m barriers. Having committed the state for the m^{th} barrier, they then start executing the "trap" request. Since it is different for all replicas, the verification at the $m + 1^{th}$ barrier will fail. Note, however, that rolling back the application and execution state to the m^{th} barrier and resuming execution sequentially is not an option, since replicas will still diverge, since the "trap" request is different for all of them. And, as we discussed above, rolling back to a state before the m^{th} barrier is also not an option.

The problem in this case arises because, by committing the first barrier of the batch, the replicas have implicitly forced to execute the entire batch, without having reached agreement on the contents of the batch. In Eve, that is acceptable, since replicas do not commit anything during the batch. Since Aegean needs to commit during the batch, however, the agreement on the batch contents must be made explicit. To that end, in Aegean replicas include the batch contents in the hash computation for the first barrier of each batch. This ensures that, if verification succeeds and the replicas mark that state as stable, the batch contents are guaranteed to be identical across all replicas.

5 The woes of sequential execution

Most replication protocols [11, 13, 15, 40, 42, 44, 47, 52] rely on the agree-execute architecture [63], where replicas first agree on the order of client requests and then execute them sequentially in that order. This approach has been the cornerstone of building replicated services, as it ensures replica consistency; i.e., it ensures that all replicas will go through the same state transitions and will produce the same output.

In a multi-service environment, however, sequential execution has undesirable consequences. Figure 2 illustrates a simple example where service A needs to make a nested request to a backend service B for every request it processes. If A is bound by sequential execution, it has no choice but to remain idle while its nested requests are being transmitted and processed. If, for example, the processing time at the backend service is approximately equal to that of the middle service, then service A will remain idle 50% of the time, leading to an inefficient implementation. This inefficiency can be further exacerbated when the message round-trip time is significant, as in geo-distributed deployments, or when more services are involved in the chain of nested requests; e.g., service B must also make a nested request to another service C. As the complexity of modern distributed systems increases, so does this chain of nested requests become longer, leading to even more inefficient implementations.

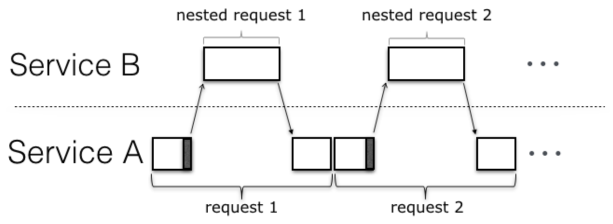


Figure 2. Sequential execution of requests in a multi-service setting. Bound by sequential execution, service A must remain idle while its nested requests are being processed at service B.

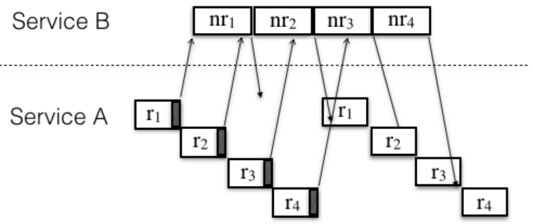


Figure 3. This figure shows how request pipelining increases efficiency, by allowing service A to keep executing requests while its nested requests are being processed at service B. This example uses a pipeline of depth 4.

5.1 Request pipelining

The insight that lets us overcome this problem is that, while sequential execution is sufficient to guarantee replica consistency, it is not necessary. In fact, any *deterministic* schedule of execution is enough to guarantee replica consistency.

With this in mind, we introduce *request pipelining*, a deterministic schedule of execution that is better suited to the multi-service environment and is free of the inefficiencies of sequential execution. The idea behind request pipelining is to allow a service to keep executing requests while its nested requests are being transmitted and processed. With request pipelining, requests are still being executed one at a time. When a request r_1 needs to make a nested request to another service, however, the replica does not remain idle waiting for the response. Instead, it starts executing the next request r_2 according to the agreed upon order of requests. Request r_2 will be executed either to completion or until that request needs to make a nested request, too. This process will be repeated k times, effectively creating a pipeline of depth k. When r_k finishes executing—partially or fully—the replica will resume the execution of r_1. At this point, it is still possible that the replica will have to wait for the response to the nested request of r_1 to arrive. If the pipeline is deep enough, the replica should be kept busy until the response arrives.

Even if the response to the nested request of r_1 arrives before it is time for r_1 to execute again, the replica will not resume the execution of r_1 prematurely. This is crucial to ensure that all replicas execute requests in a deterministic manner, as the response to the nested request of r_1 is not guaranteed to reach all replicas of A at the same time. Figure 3 illustrates an example where service A issues nested requests to service B using a pipeline of depth 4. Note that the response to the nested request of r_1 arrives before r_3 has finished executing. While it would be possible for the replica to prematurely end the pipeline and resume execution of r_1 at that point, it would be dangerous to do so, as it is not guaranteed that other replicas of service A will also receive the response at that time. Instead, request pipelining decides the order of execution based on a deterministic round-robin schedule, keeping it independent of the arrival of responses. Finally, we ensure that nested requests execute at the backend service in the order they were issued by assigning a sequence number to each nested request.

The only assumption we make in order to ensure that the schedule is deterministic is that the nested request be performed at a deterministic point during the execution of the original request. Note, however, that this should always be true for any service that is replicated using State Machine Replication, as a fundamental assumption of SMR is that the state machine executed at each replica be deterministic [56]. Similar to traditional replication protocols, we account for sources of non-determinism such as calls to *rand()* and *gettimeofday()* by having the primary assign a seed and a timestamp to each batch of requests [13, 15].

Performance benefit The performance benefit of request pipelining comes from allowing the middle service to keep processing requests, without having to wait for the responses to its nested requests. In other words, if the pipeline is deep enough, the middle service can decouple the latency of its nested requests from its throughput. As long as neither the middle service nor the backend service is saturated, increasing the depth of the pipeline will yield a performance increase. In Section 7 we describe a set of experiments that quantify the benefit of request pipelining in various settings.

5.1.1 Implementing linearizability; or not

As we discussed in Section 3, this work aims to decouple linearizability from replication. The purpose of State Machine Replication is to provide the SCM abstraction and therefore a replication protocol should provide *indistinguishability* from a single correct server. If that server is not executing requests sequentially, it would be too restrictive to require that the replication protocol *enforce* a linearizable schedule.

Request pipelining is a first attempt at freeing the replication library from the obligation to provide overly restrictive guarantees, and reaping the corresponding performance benefits. Note that the use of request pipelining *does not prevent* the service from providing linearizability. We merely apply the end-to-end argument [55] and let the application decide what kind of consistency-performance tradeoff it wants to make. The application can, for example, have its requests acquire a global lock to ensure a linearizable execution—while, of course, forfeiting the performance benefits of request pipelining. In Section 6 we elaborate on some implementation challenges raised by such lock acquisitions.

5.2 Parallel pipelining

Request pipelining is a way to prevent a middle service from remaining idle while its nested requests are being processed by the backend service. While request pipelining is a natural optimization over sequential execution, its benefits are not limited to sequential executions. Any time a service would wait for its nested requests to be serviced, we have an opportunity to optimize its efficiency by starting to execute other requests instead. In Figure 1(b), for example, the execution remains idle while verification is being performed and while the backend service processes the nested requests.

To address this inefficiency, we generalize the idea of request pipelining to parallel executions that use spec-tame. The resulting execution, which we call parallel pipelining, works as follows. When a parallelBatch of requests hits the barrier and is waiting for an external action, such as verification or backend processing, we yield execution to another parallelBatch, just as a request in request pipelining would yield execution to another request. Similar to request pipelining, the pipeline can be made k levels deep, as long as there exist enough parallelBatches in the current batch.

6 Implementation

We have implemented a prototype of Aegean, a replication library that implements all techniques put forth in this paper: *server-shim*, *response-durability*, *spec-tame*, and *request pipelining*. We implemented Aegean in Java, built upon the codebases of UpRight [15] and Eve [38], and we therefore support both the agree-execute and the execute-verify architecture. In this section we describe some important optimizations and an interesting challenge we faced during our implementation. Table 1 summarizes these techniques.

Technique	Description
Implicit agreement	Leverage the backend service to implicitly perform agreement on behalf of the middle service.
Optimize Eve	When checking for divergence in a CFT setting, ignore replica states and only compare replica outputs.
Avoid deadlocks in request pipelining	To avoid introducing deadlocks in request pipelining, synchronization operations cause a thread to yield the pipeline.

Table 1. Summary of the implementation techniques used in the Aegean prototype.

6.1 Implicit agreement

In a multi-service setting, we can leverage the backend service to implicitly perform various agreement tasks. The idea here is that the backend will refuse to act on a nested request until it has made sure that all *previous* nested responses are agreed upon and durable.

Consider, for example, the additional phase we introduced in Section 4.2 to ensure the durability of nested responses. Instead of introducing this additional phase, we: a) augment the nested requests with a recursive hash of all previous nested responses and b) require that the backend service only respond to a nested request after gathering a quorum of $max(u, r) + 1$ matching copies[3] of that request. This ensures that there are at least $u + 1$ replicas that have received the same sequence of nested responses before sending this nested request, thus ensuring the durability of nested responses.

We also leverage the backend service to perform Eve's verification implicitly. When a multithreaded execution reaches a barrier that includes sending nested requests, we don't actually send an explicit verification request to the verifier replicas. Instead, we piggyback that verification request on our nested requests and modify the backend to only act on these nested requests when it receives $max(u, r)+1$ matching copies. The only time we use an explicit verification phase is at the end of a batch or if the backend tells us that it failed to gather a quorum in a timely manner.

6.2 Optimizing Eve

Our prototype implements a simple—and yet impactful—novel optimization for Eve in the crash fault tolerant (CFT) setting. The original implementation of Eve maintains a Merkle tree with all application objects and calculates the root of the tree at the end of each batch, in order to reach agreement on the replicas' states. This results in a large hashing overhead: all responses, modified application objects, as well as all affected internal Merkle tree nodes must be hashed.

Our observation is simple: in a CFT setting, we do not need to reach agreement on the application states of the replicas; it is enough to agree on the output of the application—i.e., the outgoing nested requests and the responses to the clients. Excluding the application states from our hashing computation reduces overhead significantly, but introduces the problem of *latent divergence*. This can occur if, after executing a set of requests in parallel, all replicas reach different states but their output is still identical. Unlike Eve, we do not rollback in this case. Instead, we mark the state as committed and perform the corresponding output commit—either to the client or to another service. If later the state divergence manifests as differences in the replicas' outputs, we can simply ask the replicas to repair the divergence by performing state transfer off of an arbitrarily selected replica. This optimization only works in the CFT setting, of course; in the presence of Byzantine replicas it would not be safe to perform state transfer from another replica without assurances that its state is correct.

[3] $max(u, r) + 1$: at least $u + 1$ to ensure durability and at least $r + 1$ to avoid acting only on the nested requests of malicious replicas.

Algorithm 1 Pseudocode for acquire

```
1: procedure ACQUIRE(x)
2:     while !x.trylock() do
3:         yield()                // Hit the barrier or yield pipeline
4:         waitForMyTurn()
5:     end while
6: end procedure
```

6.3 Avoiding deadlocks

Request pipelining enforces a deterministic round-robin schedule among requests: a request yields to the next request in line when it needs to send a nested request. An undesirable side-effect of enforcing this round-robin schedule among half-finished requests is that it raises the possibility of introducing artificial deadlocks if a request holds resources—e.g., exclusive locks—when it yields. Consider the case where request r_1 acquires lock x and then yields to the thread executing request r_2. If r_2 needs to acquire lock x, as well, this leads to a deadlock: r_1 will keep waiting for r_2 to yield, while r_2 will be waiting for r_1 to release the lock.

To address this side-effect, we introduce an intermediate layer of control over thread synchronization operations. This layer implements a simple principle: instead of blocking on a synchronization operation, a thread will yield the pipeline instead. We have implemented this principle for lock acquisitions, but the idea is similar for other synchronization primitives—e.g., condition variables, semaphores, ad-hoc synchronization.

In our implementation we replace the `lock(x)` calls with calls to `acquire(x)`. Algorithm 1 shows the pseudocode for `acquire`, which makes use of Java's atomic `trylock()` operation, which acquires the lock and returns true if the lock is available; otherwise it returns false without waiting to acquire the lock.

The use of `acquire` prevents deadlocks from being introduced by ensuring that, if a lock is already acquired, no other request will be blocked waiting for that lock to be released. Instead, that request will immediately yield control of the execution to the next request.

In our current implementation we manually replace the calls to `lock` with calls to `acquire`, as it requires minimal effort for the applications we are considering. We are also considering implementing a small module that would automatically instrument the application, to eliminate the programmer effort involved.

Similar deadlocks are possible in parallel executions of Aegean, too, if requests are holding exclusive locks while performing nested requests. In that case, an unsuccessful `acquire` will cause the thread to hit the barrier.

7 Evaluation

Aegean's primary goal is to ensure the correctness of replication protocols in the presence of service interactions. Beyond correctness, however, we are also interested in the performance of the resulting replicated systems. In this section we aim to quantify the performance of Aegean-replicated services. Throughout our evaluation we will compare with the performance of existing replication protocols, to illustrate the performance benefits and overhead of Aegean. We will refer to existing replication protocols with the prefix *original*—e.g., original-Eve. Keep in mind that these protocols are actually incorrect in a multi-service setting; in our implementation we simply disregard their potential inconsistencies and report their performance as a point of reference.

Our evaluation tries to answer the following questions:

- How does the performance of Aegean-replicated services compare to existing replication protocols in a multi-service environment?
- How do various techniques and workload characteristics affect the performance of Aegean?

We answer these questions by evaluating the performance of Aegean using the TPC-W benchmark [58, 59], an established benchmark whose call pattern closely resembles that of real-world applications, such as online stores, microservices, etc. We also evaluate the performance of Aegean on a microbenchmark including two interacting replicated services. Clients send their requests to an externally facing replicated service which executes them, and half-way through execution makes a nested request to a backend replicated service. We use this setting to test how various workload characteristics affect the performance of Aegean and to determine the benefit of each of our individual techniques. The amount of computation incurred by each request in each service is configurable, with a default of 1 ms.

To demonstrate the generality of our approach, we have implemented and evaluated Aegean in a set of diverse settings. The first is a synchronous primary-backup protocol, with two replicas that use active replication and multithreaded execution. We denote the original, incorrect version of this protocol as **original-PB** and our correct and optimized version as **Aegean-PB**.

The second setting is an asynchronous replication setting in the agree-execute architecture—i.e., using sequential execution—representing Paxos and PBFT. The performance of the original, incorrect Paxos and PBFT is almost identical, so we represent them with a common line, denoted **original-sequential**. Our correct and optimized version is denoted **Aegean-CFT-singlethreaded**—again, the BFT results are almost identical and therefore omitted.

Our final setting is that of asynchronous multithreaded replication, representing Eve. Both CFT (crash fault tolerant) and BFT (Byzantine fault tolerant) modes of the original, incorrect Eve are similar and are denoted **original-Eve**. Our correct and optimized versions are denoted **Aegean-CFT-multithreaded** and **Aegean-BFT-multithreaded**; the performance of these two versions is actually different due to our optimizations (see Section 6.2).

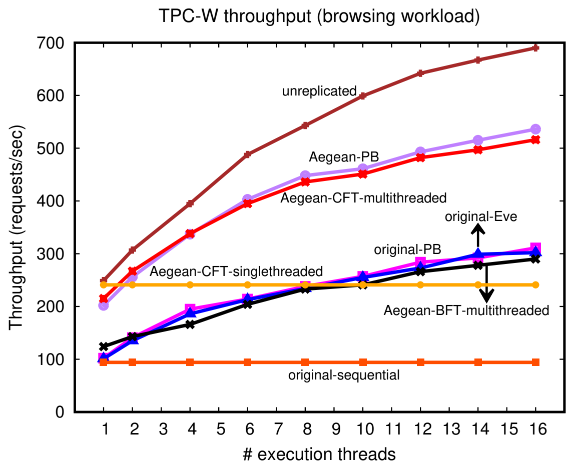


Figure 4. The performance of Aegean running the TPC-W benchmark.

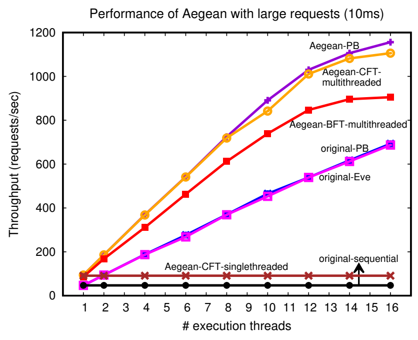
We run our experiments on a cluster of 20 nodes: 6x 16-core Intel Xeon E5-2667 v4 @3.20 GHz and 14x 4-core Intel Xeon CPU E5-1620 v4 @3.50 GHz, connected with a 10 Gb Ethernet. We use the 6 16-core machines to host our middle and backend service execution replicas. Each middle service uses a configuration that tolerates one failure—either crash or Byzantine, depending on the protocol—in each of its stages (execution, verification, or agreement) and thus has $u + max(u, r) + 1$ execution and $2u + r + 1$ verification/agreement replicas [15, 38]. The only exception is our active primary-backup mode, which always uses two replicas. Unless otherwise noted, the backend service is unreplicated.

7.1 Performance of TPC-W

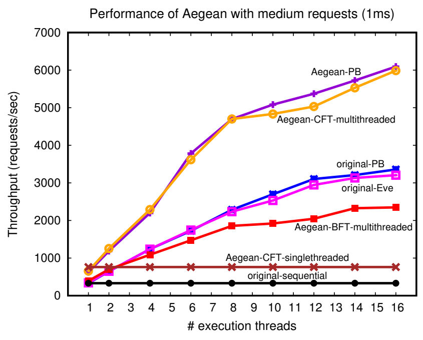
Figure 4 shows the results of our first experiment, which evaluates the performance of Aegean on the browsing workload of the TPC-W benchmark. This benchmark emulates a number of clients issuing requests to an application server (middle service). These requests affect the application server's internal state and also issue between one and three nested requests to a backend database and between zero and one nested requests to a backend payment service. We use the H2 Database Engine [30] as our database.

Figure 4 compares the throughput of various modes of Aegean with their original, incorrect counterparts. We also compare against the performance of an unreplicated application server and database, both using parallel execution.

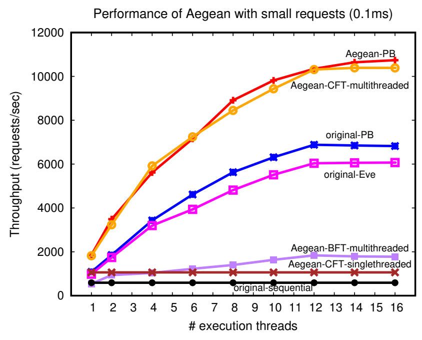
The first thing to note is that almost all Aegean modes have a clear benefit over their original counterparts thanks to request pipelining, which manages to keep both services busy most of the time, incurring a speedup of up to 2.5x. The only exception is our Aegean-BFT-multithreaded mode, which achieves almost the same throughput as original-Eve.



(a) Request duration = 10 ms.



(b) Request duration = 1 ms.



(c) Request duration = 0.1 ms.

Figure 5. The performance of Aegean running our chained services with various request durations.

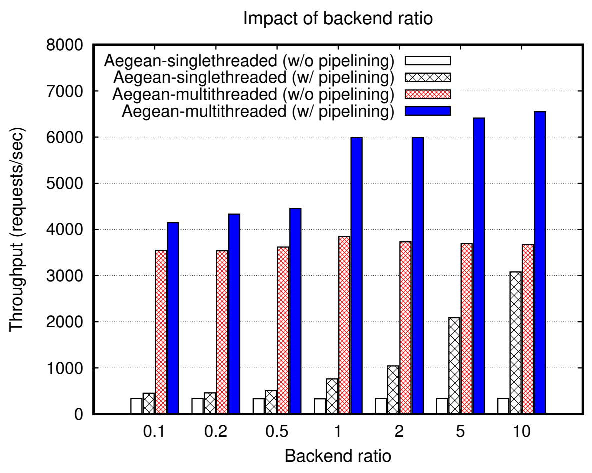


Figure 6. The effect of the processing ratio between the middle and backend service on the performance of Aegean.

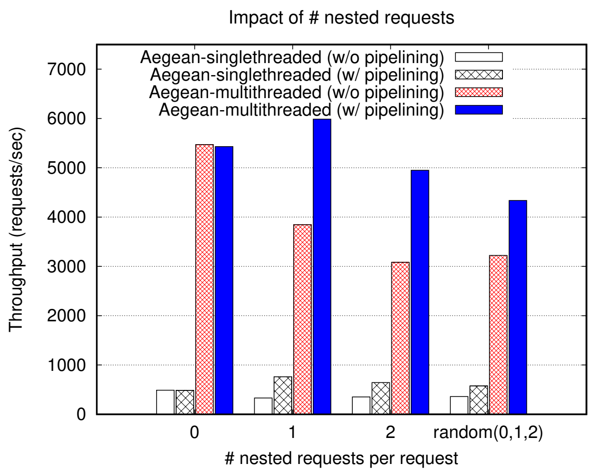


Figure 7. The performance of Aegean under various numbers of nested requests per request.

Here, the benefits of pipelining are counteracted by the overhead of having to resolve speculation—and thus perform hashing—more often than original-Eve and of having some threads wait until all threads hit the barrier. Our Aegean-CFT-multithreaded mode, on the other hand, uses our hashing optimization to drastically reduce that overhead.

Taking a step back, we want to highlight the practical importance of spec-tame: it allows Aegean to safely employ multithreaded, speculative execution, drastically improving its performance compared to sequential execution. Despite the overhead of having to resolve speculation frequently, the Aegean-CFT-multithreaded mode achieves a throughput 5.6x that of sequential execution, which compares favorably with the upper bound of 6.9x achieved in the unreplicated setting.

7.2 Effect of request size

In this section, we use our chained services to evaluate the performance of Aegean as the computation per request changes. Figure 5 shows the performance of Aegean when requests execute for 10 ms, 1 ms, and 0.1 ms, respectively, at both the middle and the backend service. Similar to our TPC-W experiment, pipelining leads to a 2x speedup in almost all modes compared to their original versions. As expected, Aegean achieves higher speedups with heavy requests since its overhead is better amortized. In most modes, however, the difference between large and small requests is not drastic. The mode that is mostly affected is again the Aegean-BFT-multithreaded mode, which has a high hashing overhead and thus benefits the most from overhead amortization.

Once again, the importance of supporting and optimizing multithreaded execution—through spec-tame and parallel pipelining—is evident: even for small requests, our Aegean-CFT-multithreaded mode achieves a 17x speedup compared to sequential execution; with large requests, this speedup increases to 25x.

7.3 Microbenchmarks

In this section, we run a set of experiments on our chained services to evaluate the impact of various workload characteristics on the performance of Aegean. In these experiments, parallel executions—pipelined or not—use 16 threads, unless otherwise noted; all services run in the CFT setting; and the default execution time is 1 ms. The backend service uses parallel execution and we vary the number of threads and use of pipelining in the middle service.

Backend ratio Figure 6 shows the results of the first experiment, which evaluates how the *backend ratio*—i.e., the ratio of computation per request at the backend service over the computation per request at the middle service—affects the performance of Aegean. The backend ratio is particularly relevant to the performance benefit of pipelining. In sequential execution a large ratio means that the middle service spends most of its time waiting for a response from the backend and thus pipelining is of particular use.

Note also that as the backend ratio increases, the single-threaded execution benefits more from pipelining than multi-threaded execution. This is because a single-threaded execution at the middle service finds it harder to reach saturation and match the throughput of the backend service. When execution at the backend is 10 times heavier than the middle, for example, pipelining yields a throughput that is 9 times higher than that achievable in traditional sequential execution.

Number of nested requests Figure 7 shows the results of the second experiment, which evaluates how the performance of Aegean is affected by the number of nested requests for every request at the middle service. We use four configurations: no nested requests; one nested request per request; two nested requests per request, and one where requests issue anywhere between zero and two nested requests. In all settings, the total computation time for a request (including all nested requests) is 2 ms, split equally among the involved services.

Observe that the impact of number of nested requests per request is significant for parallel execution. This is because the execution can hit multiple barriers in every batch, depending on how many threads execute in parallel. In contrast, when no nested requests exist, speculation need only be resolved once, at the end of the batch. As expected, in the absence of nested requests, pipelining has no effect. Adding a second nested request per request reduces the peak throughput slightly, as the overhead associated with nested requests increases. In the final configuration, where requests make a random—but still deterministic across replicas—number of nested requests, the peak throughput drops a bit further, despite the fact that the average number of nested requests is reduced compared to the previous configuration. This happens because the variability among requests is increased, causing them to hit the barrier at different times, and thus increases the time that threads wait on each other. In all these cases, however, the peak throughput of Aegean is between 14x and 18x that of sequential execution.

Longer chain Figure 8 shows the results of our final experiment, which adds a third level to our chain of services. In these experiments the second-level service is replicated and sends nested requests to the third-level service. Observe

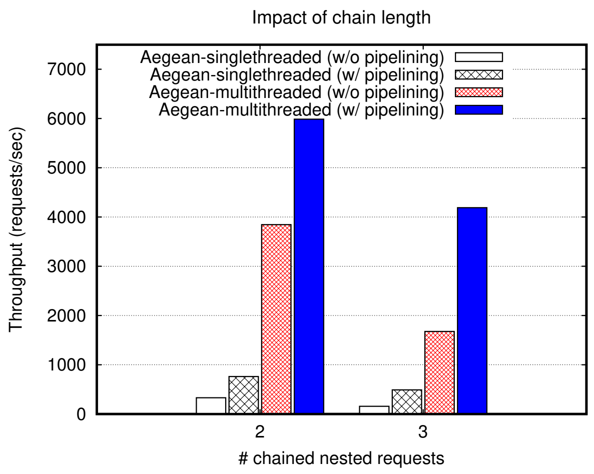


Figure 8. The performance of Aegean with longer chains of nested requests.

that the original two-level system obtains a 2.3x throughput increase over sequential execution by using pipelining. Parallel execution does not get as much of a benefit from pipelining, because it is harder to keep the pipeline full when both services have 16 threads. When we add a third layer, pipelining yields even more benefit: request pipelining is 3.15 times faster than sequential execution. In our parallel modes, we issue nested requests at the beginning of requests, to make it easier to keep all services saturated and thus demonstrate the full potential of pipelining: a 2.5x performance improvement over non-pipelined parallel execution and a 26x improvement over sequential execution.

8 Related work

In this section we review previous work on service interaction and comment on the relation to Aegean.

Replicated Remote Procedure Call Some early works provide the functionality of a replicated Remote Procedure Call [17, 18, 62]. These approaches provide mechanisms to detect and disregard duplicate copies of nested RPC calls, but do not consider complications arising from asynchrony or unreliable delivery. Most importantly, these approaches predate most replication protocols and therefore do not consider the consequences of those protocols' design choices. In particular, they do not address the performance limitations of sequential execution, and they do not consider the possibility that the RPC of one service might be based on speculative state that should not be exposed to other services.

Optimizing procedure calls among services More recently, Song et al. proposed RPC Chains [57], a technique that allows a number of interacting services to optimize complex patterns of RPCs, by composing multiple such remote calls into a single path that weaves its way through all the required services. This technique aims to reduce latency by eliminating the requirement that an RPC always returns to the caller before the next RPC is called. Although operating in a similar setting, where multiple services interact to provide some high-level functionality, Aegean and RPC Chains have very different goals. Aegean's main goal is to allow such interactions for replicated services, while RPC Chains targets singleton services.

Replicating interacting services When an interacting service needs to be replicated, currently one has to design a custom protocol tailored to this particular instance. For example, previous work proposed a custom protocol to replicate the 2PC coordinator to avoid blocking [28, 49]. In Salus [61] a replicated region server needs to issue requests to a replicated storage layer and the authors introduce a custom protocol to enable such interactions. Similarly in Farsite [3] groups of replicated nodes can issue requests to other such groups. To simplify these interactions, Farsite groups communicate through message passing and avoid using nested calls altogether. As such, it does not address the complications most services would face in such settings. Aegean, instead, tries to provide a general solution that transparently allows each layer to provide the SCM abstraction, thereby facilitating the interaction between replicated services.

9 Conclusion

In a world of large-scale systems and microservices, it becomes imperative to rethink our replication protocols to allow for interactions between multiple components. This paper puts forth a number of techniques that allow interacting services to be replicated both safely and efficiently in this brave new world.

Acknowledgements

A number of people provided feedback in the early stages of this work. We want to thank Lorenzo Alvisi and Yang Wang for their insights and David Wetterau, Jim Given, Ji Hong and Andy Ren for their contributions to earlier versions of the Aegean codebase. We also want to thank our meticulous shepherd, Lidong Zhou, and the anonymous SOSP reviewers for their detailed feedback and numerous suggestions for improving the paper. This project was funded in part by the National Science Foundation under award CSR-1814507 and by a Google Faculty Award.

References

[1] Personal communication with Google engineers, 2018.

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

[3] Adya, A., Bolosky, W., Castro, M., Chaiken, R., Cermak, G., Douceur, J., Howell, J., Lorch, J., Theimer, M., and Wattenhofer, R. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI '02*.

[4] Amazon. The amazon online store. http://www.amazon.com.

[5] Amazon. Amazon sns. https://aws.amazon.com/sns/.

[6] Amazon. Amazon sqs. https://aws.amazon.com/sqs/.

[7] Amazon. What are microservices? https://aws.amazon.com/microservices/.

[8] Ananthanarayanan, R., Basker, V., Das, S., Gupta, A., Jiang, H., Qiu, T., Reznichenko, A., Ryabkov, D., Singh, M., and Venkataraman, S. Photon: Fault-tolerant and scalable joining of continuous data streams. In *SIGMOD '13*.

[9] Baker, J., Bond, C., Corbett, J. C., Furman, J., Khorlin, A., Larson, J., Leon, J.-M., Li, Y., Lloyd, A., and Yushprakh, V. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)* (2011), pp. 223–234.

[10] Balakrishnan, M., Malkhi, D., Davis, J. D., Prabhakaran, V., Wei, M., and Wobber, T. Corfu: A distributed shared log. *ACM Trans. Comput. Syst. 31*, 4 (Dec. 2013), 10:1–10:24.

[11] Balakrishnan, M., Malkhi, D., Prabhakaran, V., Wobber, T., Wei, M., and Davis, J. D. Corfu: A shared log design for flash clusters. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 1–1.

[12] Budhiraja, N., Marzullo, K., Schneider, F. B., and Toueg, S. Primary-backup protocols: Lower bounds and optimal implementations. In *CDCCA* (1992).

[13] Castro, M., and Liskov, B. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.* (2002).

[14] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association.

[15] Clement, A., Kapritsos, M., Lee, S., Wang, Y., Alvisi, L., Dahlin, M., and Riche, T. UpRight cluster services. In *SOSP* (2009).

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

[17] Cooper, E. C. Replicated procedure call. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 1984), PODC '84, ACM, pp. 220–232.

[18] Cooper, E. C. Replicated distributed programs. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 1985), SOSP '85, ACM, pp. 63–78.

[19] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst. 31*, 3 (Aug. 2013), 8:1–8:22.

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

[21] Cui, H., Gu, R., Liu, C., Chen, T., and Yang, J. Paxos made transparent. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 105–120.

[22] Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N., and Warfield, A. Remus: High availability via asynchronous virtual machine replication. In *NSDI* (2008).

[23] Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[24] Decandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *SOSP* (2007).

[25] Expedia. The expedia travel metasearch engine. http://www.expedia.com.

[26] Gafni, E., and Lamport, L. Disk paxos. *Distrib. Comput. 16*, 1 (Feb. 2003), 1–20.

[27] Google. Microservices architecture on google app engine. https://cloud.google.com/appengine/docs/standard/python/microservices-on-app-engine.

[28] Gray, J., and Lamport, L. Consensus on transaction commit. *ACM Trans. Database Syst. 31*, 1 (Mar. 2006), 133–160.

[29] Guo, Z., Hong, C., Yang, M., Zhou, D., Zhou, L., and Zhuang, L. Rex: Replication at the speed of multi-core. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 11:1–11:14.

[30] H2. The H2 home page. http://www.h2database.com.

[31] Hadoop. Hadoop. http://hadoop.apache.org/core/.

[32] HBase. The hBase homepage. http://www.hbase.org.

[33] Herlihy, M. P., and Wing, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (1990), 463–492.

[34] Hoff, T. Deep lessons from google and ebay on building ecosystems of microservices. http://highscalability.com/blog/2015/12/1/deep-lessons-from-google-and-ebay-on-building-ecosystems-of.html.

[35] Hoff, T. Lessons learned from scaling uber to 2000 engineers, 1000 services, and 8000 git repositories. http://highscalability.com/blog/2016/10/12/lessons-learned-from-scaling-uber-to-2000-engineers-1000-ser.html.

[36] Howard, H., Malkhi, D., and Spiegelman, A. Flexible paxos: Quorum intersection revisited. In *OPODIS* (2016).

[37] Junqueira, F. P., Kelly, I., and Reed, B. Durability with bookkeeper. *SIGOPS Oper. Syst. Rev. 47*, 1 (Jan. 2013), 9–15.

[38] Kapritsos, M., Wang, Y., Quema, V., Clement, A., Alvisi, L., and Dahlin, M. All about eve: Execute-Verify replication for Multi-Core servers. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation* (2012).

[39] Kayak. The kayak travel metasearch engine. http://www.kayak.com.

[40] Kotla, R., Clement, A., Wong, E., Alvisi, L., and Dahlin, M. Zyzzyva: Speculative byzantine fault tolerance. *Commun. ACM 51*, 11 (Nov. 2008), 86–95.

[41] Kreps, J., Narkhede, N., and Rao, J. Kafka: a distributed messaging system for log processing. In *Proceedings of the 6th International Workshop on Networking Meets Databases (NetDB)* (2011).

[42] Lamport, L. The part-time parliament. *ACM Trans. Comput. Syst.* (1998).

[43] Lamport, L. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32*, 4 (Dec. 2001), 51–58.

[44] Lamport, L. Fast paxos. *Distributed Computing 19*, 2 (Oct 2006), 79–103.

[45] Lamport, L., and Masa, M. Cheap paxos. In *Proc. DSN-2004* (June 2004), pp. 307–314.

[46] Liskov, B., and Cowling, J. Viewstamped replication revisited. Tech. Rep. MIT-CSAIL-TR-2012-021, MIT, July 2012.

[47] Liu, S., Viotti, P., Cachin, C., Quéma, V., and Vukolic, M. Xft: Practical fault tolerance beyond crashes. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2016), OSDI'16, USENIX Association, pp. 485–500.

[48] Mao, Y., Junqueira, F. P., and Marzullo, K. Mencius: building efficient replicated state machines for WANs. In *OSDI* (2008).

[49] Mohan, C., Strong, R., and Finkelstein, S. Method for distributed transaction commit and recovery using byzantine agreement within clusters of processors. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 1983), PODC '83, ACM, pp. 89–103.

[50] Moraru, I., Andersen, D. G., and Kaminsky, M. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2013), SOSP '13, ACM, pp. 358–372.

[51] Nightingale, E. B., Veeraraghavan, K., Chen, P. M., and Flinn, J. Rethink the sync. In *Proc. 7th OSDI* (Nov. 2006).

[52] Ongaro, D., and Ousterhout, J. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2014), USENIX ATC'14, USENIX Association, pp. 305–320.

[53] Reinhold, E. Rewriting uber engineering: The opportunities microservices provide. https://eng.uber.com/building-tincup/.

[54] Riady, Y. Distributed sagas for microservices. https://yos.io/2017/10/30/distributed-sagas/.

[55] Saltzer, J., Reed, D., and Clark, D. End-to-end arguments in system design. *ACM Trans. Comput. Syst. 2*, 4 (1984), 277–288.

[56] Schneider, F. B. Implementing fault–tolerant services using the state machine approach: A tutorial. *Computing Surveys 1990.*

[57] Song, Y. J., Aguilera, M. K., Kotla, R., and Malkhi, D. Rpc chains: Efficient client-server communication in geodistributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09).*

[58] TPC-W. Open-source TPC-W implementation. http://pharm.ece.wisc.edu/tpcw.shtml.

[59] Transaction Processing Performance Council. The TPC-W home page. http://www.tpc.org/tpcw.

[60] Wang, Y., Alvisi, L., and Dahlin, M. Gnothi: Separating data and metadata for efficient and available storage replication. In *USENIX ATC* (2012).

[61] Wang, Y., Kapritsos, M., Ren, Z. A., Mahajan, P., Kirubanandam, J., Alvisi, L., and Dahlin, M. Robustness in the salus scalable block store. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)* (2013).

[62] Yap, K. S., Jalote, P., and Tripathi, S. Fault tolerant remote procedure call. In *Proceedings of the 8th International Conference on Distributed Computing Systems* (Jun 1988), pp. 48–54.

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