Promises: Enabling Consistent, Mutable, Popular Objects for Reliable Distributed Systems

Michael George (mdgeorge@cs.cornell.edu)

Special Committee:

Andrew Myers (chair)	Fred Schneider	Hakim Weatherspoon	Lawrence Gibbons
andru@cs.cornell.edu	fbs@cs.cornell.edu	hweather@cs.cornell.edu	lkg5@cornell.edu

October 8, 2009

Abstract

When building distributed systems that manage mutable objects, there is a fundamental tension between scalability and consistency: strong consistency requires tight coordination, while scalability requires decoupling. Unfortunately, many distributed applications require some degree of mutability, consistency, and scalability. In particular, these requirements are crucial for building reliable applications in critical areas such as finance and health care.

In the proposed research, we aim to support the large class of applications that infrequently update their popular objects. We introduce promises, a mechanism that allows these applications to dynamically and flexibly trade off mutability and scalability for individual objects. We will show that promises enable a wide variety of critical applications that require consistency, mutability and scalability.

1 Introduction

Emerging Internet services have enabled new and exciting applications that integrate large amounts of information from diverse sources. Users can search the entire web through Google, view a vast collection of videos through YouTube, or manage social networks using Facebook or MySpace. Moreover, these sources can be integrated to produce mashups, placing photos from Flickr onto maps from MapQuest, or pulling together airline rates from Orbitz and combining them with hotel rates from Hotels.com.

Unfortunately, this kind of data integration has failed to penetrate critical areas such as finance and health care. Even a simple task like detecting potential drug interactions requires a person to manually contact the practitioners involved and gather prescription information from separate patient charts. This process is costly and error prone, and the results are deadly: according to a 1999 Institute of Medicine study, at least 44,000 deaths annually result from medical errors, with incomplete patient information identified as a leading cause [**?**].

Underlying these shortcomings is a lack of strong consistency guarantees in large scale distributed systems. Applications need to be able to perform multiple reads and updates in atomic transactions that are executed as if they were on a reliable central system. Without consistency enforcement, updates to critical data can be lost, or computations can produce incorrect values.

Consistency is not the only requirement for integrated, reliable applications. Such applications also need to enforce security policies on the information they manage, and maintain invariants over

distributed data. Without consistent transactions, however, systems can provide little assurance that security policies are correctly enforced or that invariants are maintained.

Unfortunately, there is a fundamental tension between consistency, mutability and scalability. Systems can easily provide any two of these properties, but providing all three is impossible. Supporting consistent transactions against objects that can change over time requires some form of centralized coordination, but scalability requires decentralization. Large distributed systems must eschew consistency or mutability in order to achieve sufficient scalability.

Our insight is that while many important applications require consistency, their scalability is constrained by popular data that is mutable but updated infrequently. My research introduces promises, a mechanism that allows applications maintain consistency while flexibly trading mutability for scalability.

I will demonstrate that promises enable a wide variety of reliable and scalable applications. I will implement and evaluate promises in the context of Fabric [**?**], a system developed by the Applied Programming Languages group at Cornell to deliver on the promise of secure, reliable, and large scale internet applications.

This proposal is organized as follows: Section 2 defines terms and explains assumptions that we will rely on throughout the remainder. Section 3 describes the promises mechanism in detail and shows how it enables a scalable "zero-phase commit" for popular objects. Section 4 discusses the challenges involved in programming against a system with promises, and outlines our proposed language extensions for enabling this. In section 5, I lay out a plan for the completion of the proposed research. Section 6 discusses related work, and section 7 concludes.

2 System Assumptions

We refer to the units of data in a system as *objects*. We assume that objects are small, but that they can refer to other objects and form larger data structures. Note that objects are an abstraction of data in many types of systems: database rows, i-nodes, and log entries are all examples of objects. In general, objects can refer to other objects managed by different stores, and transactions can span objects that are located at different stores.

A distributed system is partitioned across network nodes. Each object must have some node responsible for storing the definitive current version and for ensuring consistent access to it. We refer to this node as the object's *store*. For fault tolerance, stores may be replicated, but because we require consistency, they can only be replicated across a small group of tightly coupled machines. Thus we will think of objects as being located at and managed by a single node.

We assume that objects are read, created, and modified in transactions that satisfy the ACID properties: either all of the effects of a transaction happen, or none do (atomicity), they bring the system from an acceptable state to an acceptable state (consistency), they do not interfere with the execution of other transactions (isolation), and once complete, they remain complete (durability). We will refer to the nodes that execute transactions as *workers*.

In order to prevent stores from becoming bottlenecks, many systems have some kind of *dissemination cache*. The role of the dissemination cache is to allow clients to read objects without contacting the store directly. Because the cache must be widely distributed to be effective, it cannot provide strong consistency guarantees; instead we assume it makes a "best effort" to provide relatively up-to-date copies of objects. Thus the workers and stores must coordinate to ensure that transactions are committed consistently.

Because of the fundamental tension between consistency, mutability and scalability, we can not support transactions that access objects in arbitrary ways. Instead, we restrict our focus to a class of applications that we believe encompasses a large number of important applications. In particular, we allow applications to have mutable *popular objects*, but we assume that updates to popular objects are infrequent.

For example, consider a simple directory service that contains a root object which points to home directory objects for a large number of users (figure 1). The common operations against the directory

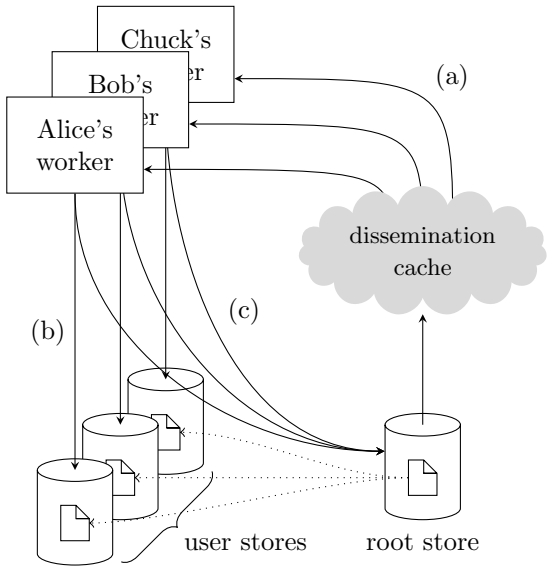


Figure 1: File directory example. There is a root object on the root store that points to user directory objects on the user stores. (a) The workers read data from the dissemination cache. (b) Workers commit updates back to the stores. (c) Workers must coordinate with the root store to ensure that the version of the root object that they read is still consistent with the modifications they make.

would be to look up a single home directory and read or modify its contents, but the service also needs to support occasional modifications to the root directory itself, for example to add or remove a user. The root directory is an example of a popular object: it is read by every transaction that uses the service, and it is mutable but infrequently updated.

This example clearly shows the tension between scalability and consistency. The dissemination cache is scalable because it does not require consistency. The non-popular objects are scalable because they can be partitioned across stores. The bottlenecks in the system are clearly the stores holding popular objects: every request reading a popular object must coordinate with the store to ensure that the object has not changed.

3 Promises

A key insight is that in order to allow consistent access to popular objects, the worker must be certain about the state of the popular objects it has read without contacting the corresponding stores. To make this possible, stores that hold popular objects may issue *promises*. A promise is a predicate over the state of the objects located at a store that is guaranteed to hold until a certain time.

For example, the root store in figure 1 may issue a promise stating that the value of the root directory is fixed until a given time. Promises of the form "$o = v$ until t" are the simplest form of promise; we refer to them as *equality promises*.

Alternatively, the root store may issue a more specific promise about the state of the root object. For example, it may promise that the root directory object contains a reference to Alice's home directory, or that it contains at least a certain number of files. We discuss these general *semantic promises* further in section 4.

As a worker executes a transaction, it reads promises from the dissemination cache and records its updates in its local cache. When it completes its processing, it is ready to commit. Ideally, all of the promises on objects it has read are up-to date, and there are no current promises against the objects it has modified. In this case, the worker can immediately commit the transaction to the updated stores without contacting the stores from which it has read. The promises ensure that objects it has read are still consistent with the updates it is making. In this case we say we have enabled a *zero-phase commit* against those stores.

There are a number of ways to deviate from this ideal situation, some of which are depicted in figure 2. One possible problem is that promises expire, either before they reach the worker, or while the worker is processing. For example, in figure 2(b), s_f generates a promise on o_f that is expired by the time it reaches the worker. As shown in figure 2(d), after choosing a commit time for the transaction the worker must fetch up-to date versions of the promises it has acquired. It may fetch these from the dissemination cache or from the stores directly.

Figure 2: Execution of a transaction in the file directory example, with promises. The worker looks up Alice's home directory o_a in the root directory object o_r. It then copies file o_f into o_a. For illustration, we assume o_a had an outstanding promise until time t_a. The objects o_f, o_a, o_r are stored at stores s_f, s_a, and s_r respectively.

(a) The root directory o_r is popular, so s_r generates a promise "$o_r = \ldots$ until t_r" and adds it to the dissemination cache.

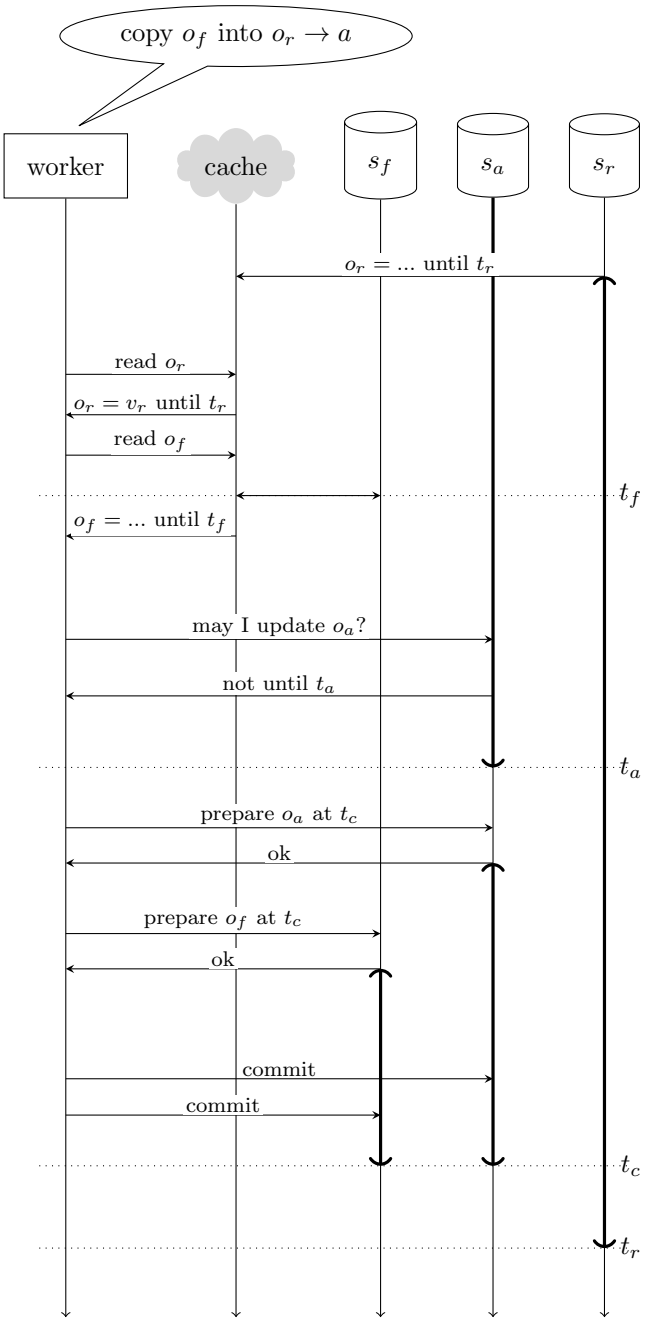
(b) The worker reads o_r and o_f from the dissemination cache. It receives the promise for o_r, but misses on o_f. The cache then contacts s_f, which gives a promise for o_f with zero duration. The worker also updates its local copy of o_a.

(c) The worker completes the transaction and begins the commit process. Since it is updating o_a, it proposes its updates to s_a. s_a has an outstanding promise against o_a until t_a, so it responds "not until t_a."

(d) The worker chooses a commit time t_c. It sends a prepare message to s_a asking whether it is safe commit o_a at t_c. In addition, since it does not have a promise about o_f that lasts through t_c, it contacts s_f to acquire one. It does not need to contact s_r because the promise it acquired about o_r is still valid.

(e) After receiving affirmative responses to all prepare requests, the worker sends a commit message. o_a and o_f may be modified after t_c.

Note that no messages were sent to t_r through this process. This example shows how promises enabled a zero-phase commit against the popular object o_r.



Another possible problem is that the updates the worker wishes to perform might violate outstanding promises. In figure 2(c), for example, the updates to Alice's home directory o_a are prevented by an outstanding promise. To deal with this situation, workers propose their updates to the stores as the first step of the commit process. If the stores have outstanding promises, they may respond with a future time at which the updates may be accepted. The worker must then choose a commit time that is later than all of these times.

To avoid preparing transactions in the past, the worker needs to choose a commit time that is likely to be after the stores have received the prepare message. Messaging delays or failures may cause the commit time to come too early. However, in this case the client can simply retry the prepare with a longer estimate for the transmission time.

As described in section 5, modeling this protocol and formally verifying that it correctly handles these exceptional circumstances is one of the early steps in the proposed research.

Promises are a generalization of both optimistic and pessimistic concurrency control. With optimistic concurrency control, workers essentially receive promises of zero duration. At commit time they are forced to send prepare messages to all stores to enforce consistency. With pessimistic concurrency control, workers receive promises of infinite duration that must be explicitly released. In addition to removing the bottleneck of communicating with popular stores, promises provide a way to dynamically adapt between the advantages of optimistic and pessimistic concurrency control.

We have left unspecified various details that will have an effect on the efficient functioning of a system supporting promises. What promises will be issued and for what durations? Under what circumstances should the worker contact the store directly to refresh expired promises? How should the dissemination cache manage expired promises? As described in section 5, determining appropriate heuristics for these decisions is an important goal of the proposed research.

4 Semantic Promises

Equality promises are appealing from a system implementation point of view because of their simplicity. They do not require the system understand the semantics of objects, and thus provide a clean separation between the system layer and the application layer.

Unfortunately, equality promises are insufficient for many applications. To see this, consider once more the file directory example in section 2. Because the root object must refer to an arbitrarily large number of user directories, it cannot be implemented as a single object. Rather, it is likely to be implemented as a data structure composed of smaller objects. For example, it might be implemented as a B-tree made up internally of tree nodes.

With equality promises, a worker that is looking up Alice's home directory would need to acquire a promise about each internal tree node it accesses as it searches through the B-tree. Although the worker is only concerned with the path to Alice's directory, these promises would prevent any concurrent updates to other paths that happen to overlap. In addition, they would make it impossible to re-balance the tree.

What the worker really needs is a semantic property of the root directory, namely that $root.lookup(\texttt{"/Alice"}) = o_a$. By acquiring a promise of this property, the worker can perform its computation while allowing other workers to modify other parts of the root, or re-balance the tree. We refer to promises of semantic properties like these as *semantic promises*.

In addition to providing greater concurrency, semantic promises are *composable* in the sense that they allow higher level abstractions to be built out of lower level abstractions. For example, the promise $root.lookup(\texttt{"Alice"}) = o_a$ is implied by the lower level equality promises about the nodes making up the B-tree. However, the store can issue the weaker semantic promise with a longer duration.

Semantic promises also enable a form of shared memoization. Workers can assemble lower level promises into semantic promises while they compute, and add them back to the dissemination cache to save other workers from performing the same computa-

tions. Unlike semantic promises issued by the store, however, these promises expire as soon as any of the lower level promises does.

4.1 Language Features for Promises

Programming with semantic promises requires support from the language used to express objects and transactions. Instead of simply reading objects, transactions must specify the semantic properties that they care about so that semantic promises can be fetched. Objects are no longer opaque blocks of data, rather they must specify what semantic promises to issue, and determine whether incoming updates violate those promises. In this section we sketch language features we are exploring for enabling these semantics to be specified. These mechanisms are still early in the design phase; a full language design and implementation is a critical piece of this proposed research.

The programs that implement transactions need a way to specify exactly the promises they are interested in. For this purpose, we introduce the `assume` statement. For example, Alice's worker in the file directory may execute the statement:

```
assume root.lookup("/Alice") == File f;
```

Here `root.lookup("/Alice") == File f` is a *promise pattern*, which is a boolean predicate with some bound variables (`root` in this example), and some unbound variables (`f` in this case).

The effect of this statement is to acquire a promise that matches the pattern and to bind the unbound variables. In the example above, the worker might fetch a promise that says that `root.lookup("/Alice") ==` o_a. Then it will assign the value o_a to the variable `f` and proceed.

The worker may alternatively execute the promise pattern using bidirectional computation [**?**] and relying on lower-level promises. In this case, it both the lower-level promises and the higher-level promise in its log, allowing it to replace the lower-level promises with the higher-level promise at commit time.

In order to avoid theorem proving at runtime, we must constrain the space of possible promises. We propose that abstraction interface specifies what promises it is capable of making. For example, the interface to the file directory may specify that it can offer promises of the form `this.lookup(String s) == File f`

5 Status and Plan of Work

This section outlines our preliminary results, and contains a detailed plan for the completion this thesis, with estimated dates.

5.1 Preliminary Work

Prior to preparing this A exam, I have taken significant steps that will enable me to investigate promises. I am one of the primary researchers on the Fabric system and language. Fabric is designed for building secure distributed information systems. It that allows heterogeneous network nodes to securely share both information and computation resources despite mutual distrust.

Building promises into Fabric will allow me to focus on the design and tradeoffs of the promises mechanism without having to build and tune the other abstractions that Fabric provides. In addition, we have implemented a number of realistic applications on top of Fabric, including a port of Cornell's Course Management System (CMS), and a secure distributed multi-user calendar. These applications will allow me to evaluate the impact of promises on real applications.

I have also built a simple mockup of the file directory example in section 2, and am running experiments to demonstrate the impact of two-phase commit on system scalability. This data will serve as a baseline against which to evaluate promises.

5.2 Equality Promises

The next phase of my research on promises will be to implement and evaluate equality promises, and to develop the heuristics required for promise generation, storage, and reaquisition. This phase will demonstrate that in certain situations, limiting the write availability of popular objects can allow consistent

systems to scale as well as inconsistent systems. In addition, we will model and formally verify that the promises protocol outlined in section 3 correctly enforces consistency.

As described in section 4, equality promises are not sufficiently flexible to handle all popular objects. Thus our goal for this phase is to show that equality promises enable high scalability for the situations where they are suitable, without negatively impacting the performance of situations where they are not. We will show this by measuring the performance of sample applications with promises, and comparing those measurements to measurements without promises, and with read-only commits disabled.

Developing good heuristics will require a combination of analysis and experimentation. In addition to implementing the necessary protocol changes to Fabric to support promises, we will implement a variety of strategies for generating promises. We will evaluate the effectiveness of these strategies by examining the performance impact on isolated test cases, and use the large Fabric applications we have developed to evaluate the broader performance implications.

5.3 Semantic Promises Evaluation

After showing that equality promises can increase system scalability in limited examples, the next phase of my research will show that this scalability win can be realized in a much more general way with semantic promises.

We will begin by completing the design of the language features sketched in secion 4. We will implement these features as an extension to the Fabric language, which is itself an extension to the Jif programming language, which in turn extends Java.

Effectively implementing semantic promises presents a number of challenges at the system level, beyond those required by equality promises. The management of promises at the store becomes much more complicated because there is a many-to-many relationship between promises and objects: a promise can encode facts about many objects, and each object may have many different promises against it.

Semantic promises also present challenges for the dissemination layer. Rather than simply mapping object identifiers to equality promises, the dissemination layer must now respond to queries in the form of promise patterns. Designing and implementing such a distributed cache will be another contribution of this phase.

My aim in this phase is to demonstrate that realistic applications will benefit from semantic promises. To evaluate this hypothesis, I will build upon my experience with equality promises to design and implement semantic promises within Fabric. In addition, I will make the necessary changes to the Fabric implementation of CMS to allow it to distributed, and will demonstrate that semantic promises allow it to scale well beyond its current limits.

5.4 Other Possible Directions

Promises present a number of interesting directions for further research. Here I list some of these directions. Although I do not have specific plans to tackle these issues, I may address some of them while working on this thesis.

Extended Promises It is possible to extend semantic promises even further by allowing them to incorporate time more directly. This would allow stores to issue promises that degrade predictably over time for values that vary continuously at a predictable rate. This may be useful for applications such as collision detection in online games.

Another possible extension of promises is to allow them to discuss objects at multiple stores. This would require coordination between the stores, but may be useful in supporting features such as object migration.

Security Properties One of the important requirements for systems like Fabric that are distributed over multiple trust domains is strong security assurance. Promises may have have implications for this: they may leak confidential data or allow malicious parties to affect trusted data. Determining the extent of these interactions and designing mechanisms to ameliorate them remains an open problem.

Weak Consistency Promises are a mechanism that enforce strong consistency but allow the system to trade off between read and write availability. Although this decision is suitable for many applications, some applications may require high read and write availability, and may be able to sacrifice consistency to achieve it. Enabling such applications would require modifications to the promises mechanism.

6 Related Work

Many distributed systems have had to grapple with the fundamental tension involved in supporting decentralized, consistent reads of mutable objects. Some systems, such as Chord [**?**], SFS/RO [**?**], Dryad [**?**] and Map/Reduce [**?**] avoid the problem by only supporting immutable objects. Beehive [**?**] in particular addresses scalable handling of read-only popular objects. Other systems give up on strong consistency and support some form of "best-effort" consistency [**?**, **?**, **?**], or restricted consistency guarantees that only apply to a single object or store [**?**, **?**, **?**].

Other systems support consistent, mutable objects, but fail to support popular objects because they require centralized coordination for reads [**?**, **?**, **?**, **?**]. There has been much work on reducing the overhead of optimistic concurrency in this setting [**?**, **?**], but none of these focus specifically on removing the bottleneck caused by popular objects.

Because they make a fixed tradeoff between consistency, mutability, and decentralized reads, none of these systems support mutable, consistent, and popular objects. As far as we know, promises are the only mechanism that supports all of these goals.

Semantic promises are similar to mechanisms in Galois [**?**] and Conits [**?**], which attempt to provide greater parallelism in distributed supporting semantic consistency. Because the focus is on greater parallelism, these systems have not addressed the scalability challenges that promises address. In addition, these systems provide increased concurrency for a handful of provided data structures, whereas promises aim to be a tool that programmers can use to build their own data structures.

Promises are similar to many other ideas in programming languages. Semantic promises are a form of memoization, and draw from work on automatic memoization of imperative programs [**?**]. Our proposed language for promise patterns was inspired by JMatch [**?**]. Because they ship facts, rather than objects, promises are also similar to distributed proof systems that have been developed by the security community [**?**, **?**].

7 Conclusion

We believe that concurrency control may become a bottleneck for popular objects in systems using optimistic concurrency control. We have proposed two mechanisms to this problem: a zero-phase commit protocol that uses loosely synchronized clocks to maintain consistency without touching the store, and a promise mechanism for flexibly specifying concurrency control. We have laid out an implementation plan that explores the performance of these mechanisms on a realistic workload using Cornell's Course Management System, which we hope will show that promises are a useful and feasible mechanism for managing popular objects.