PROJECT SUMMARY

Overview:

The general thesis of this work is straightforward: it is possible to leverage expected user behaviors in dynamic, mobile environments to support much stronger consistency guarantees, and much better performance, than has heretofore been thought possible.

We propose to design and build three versions of a general object store system that will provide strong properties for replicated data in our target environments. The base of the systems will consist of a set of simple replicating data servers that can tolerate and exploit device and network heterogeneity. Above these are versioned object stores, which manipulate high-level abstractions of object accesses and can display customized views, tailored to device characteristics. Above this is the application layer, where we will build file systems, relational databases, and simple object stores.

Keywords: distributed systems; consistency protocols; consensus protocols; data sharing.

Intellectual Merit :

To realize the plan above we will need to make the following original contributions:

1) Definition, validation, and parameterization of a user model corresponding to the important case of individual users and small collaborating groups, accessing data episodically, across a heterogeneous group of devices.

2) Definition of a new system structure that allows the underlying replication stratum to be independent of the upper levels that maintain higher-level invariants, such as strong consistency, safety, and liveness properties.

3) Definition of protocols to identify and use high-level abstractions of object accesses in optimizing performance, guiding consistency protocols, and defining object store views.

4) Definition and analyses of a set of optimizations to tailor standard consensus protocols to our target environment.

Broader Impacts :

If successful, we feel that the proposed research will have substantial real-world impact. Our target environment has been largely ignored by recent work on supporting strong consistency in distributed environments. Yet the potential audience for artifacts resulting from this work include everyone who uses Dropbox, or other stand-alone tools for replicating work across multiple devices.
Furthermore, this work will be folded into classes at both the undergraduate and graduate level, as well as included in lectures presented to high school and middle school students.

TABLE OF CONTENTS

For font size and page formatting specifications, see GPG section II.B.2.

*Proposers may select any numbering mechanism for the proposal. The entire proposal however, must be paginated. Complete both columns only if the proposal is numbered consecutively.

1 Introduction

In the grand tradition of systems researchers eating their own dogfood [91], the work described in this proposal has grown out of frustration that colleagues and I have felt about the tools we use daily. We use DropBox [3] heavily, but we want stronger consistency, durability, and performance properties [106]. In short, we wish to build a single, versioned data repository to hold all of our data, regardless of source or intended use. We can distill our desired attributes into the following six:

1. *complete* - The system should hold all data that we use, whether for work, home, in file systems, in databases. All versions of all data should be held permanently.

2. *consistent* - Data accesses should reflect the strongest consistency guarantees, regardless of the characteristics of the underlying devices and services.

3. *controlled* - The system should be entirely under user control; there should be no danger of cloud vendor lock-in.

4. *available* - All data should be transparently replicated everywhere needed, and data should be available on all devices.

5. *fast* - Local data accesses should be fast. This means limiting network data accesses, especially those off the local area network. This has nothing to do with scaling to large numbers of users, which is explicitly not a goal of this work.

6. *secure* - There should be no reliance on any company's security mechanisms and procedures, or lack thereof. The strongest cryptographic primitives should be used throughout.

None of this is an indictment of current tools and services. In fact, the characteristics we want are clearly inappropriate for many customers of current cloud services, for example. Instead, it is a clear example of the end-to-end argument [89]: our goals and priorities do not match those of the device manufacturers and service providers, so we will layer them on top of the provided systems.

Consistency is often treated as the poor step-child of the distributed computing world. Until recently, the consistency of wide-area systems was either best-effort or supported only eventual consistency, as availability is generally not compatible with strong consistency [37]. Best-effort was often augmented with service guarantees about recency, which does not map back to guarantees about consistency. Eventual consistency makes guarantees only about the final state of quiescent data; meanwhile data values can diverge across replicas, leading to necessarily inconsistent data being served to applications. Some applications can tolerate weaker consistency [25], and leading practitioners have argued that the research community's emphasis on consistency was misplaced, that scalability was the only worthwhile goal [17].

However, Google's AWS services has consistently received complaints from users that Bigtable [22] can be difficult to use for applications that want both strong consistency and wide-area replication [23, 100]. Further, the last five years have seen several systems built to support the stronger causal consistency, the strongest consistency available during partitions [71], in the wide area [69, 70, 97]. Finally, some recent commercial systems are starting to support stronger consistency. The most general of these systems is Google's Spanner [23], a global-scale system that supports linearizable transactions. However, Spanner benefits from assumptions on its environment: secure data centers with low turnover and known latencies, and a global clock infrastructure (TrueTime) synchronized at a fine granularity. These new systems are a step forward, but are designed for the data center and rely on data center-like failure, connectivity, and group membership properties.

We propose to investigate extending these strong guarantees to systems outside the data center. Our target environments include single users operating across multiple devices, small collaborating groups, and

ad hoc collections of users and devices. These environments differ from data centers in several ways, but we concentrate on three. First, they are less reliable. This makes it more difficult to build reliable protocols on top of them, but also makes reliable protocols more important. Second, they are dynamic, the set of devices a user operates will change over time, and small collaborating groups can be ephemeral or frequently change members. Finally, usage on such systems is often episodic. A single user might access data on many devices, but usually only on a single device at a time. Likewise, members of collaborating groups will often work on distinct portions of the data, or "time-slice" over specific portions of their shared data. The first two differences make designing reliable protocols more difficult, but the last one potentially provides a means to design efficient protocols.

We propose to design and build three versions of a general object store system that will provide strong properties for replicated data in our target environments. The first, Flow, will provide secure access to causally-consistent, opaque *blobs* across loosely-connected data stores. This consistency level is as strong as that of almost all distributed systems in data centers, and stronger than guarantees provided to local file systems. However, we describe a new approach to efficiently supporting causal consistency in a flexible system that separates replication from consistency constraints. This separation allows data replication to take advantage of local optimizations allowed by topological independence [14, 103], such as laptops being carried from home to work [82], or asynchronous trickle reintegration [77, 50] without affecting emergent high-level guarantees. Flow will be able to support these guarantees purely through local operations.

Flow-l will build on Flow by using the higher-level abstraction of versioned objects to inform operation of the underlying system. In particular, we will define high-level object views that allow us to take advantage of object access histories to provide better performance and stronger consistency guarantees, including linearizability during failure-free operation.

Finally, Flow-p will build on the above base to make safety and liveness guarantees by implementing full-on generalized consensus protocols. Consensus protocols will enable the system to implement strong consistency models, such as linearizability, and maintain strong durability properties. We describe several optimizations, including adaptive approaches to consensus group design, as well as the use of domain-specific relaxations of consensus assumptions and requirements.

The core of the new systems will be a set of dynamic algorithms that provide efficient support for strong consistencies (causal consistency, linearizability), and strong notions of correctness across a variety of devices. The algorithms will tolerate rapidly changing set of devices, connectivities, sharing arrangements outside the data center. The protocols will conform their actions to the set of devices involved in the decision-making processes to active devices, and move those decision-making processes closer to where the data is used.

The general thesis of this work is straightforward: it is possible to leverage expected user behaviors in dynamic, mobile environments to support much stronger consistency guarantees, and much better performance, than has heretofore been thought possible. To support this goal we will need to i) conduct user studies, ii) design protocol variants optimized for our expected usage, and iii) build and evaluate prototypes at different points in the performance/consistency space.

The remainder of this proposal is structured as follows. Section 2 will discuss the background of our problem, and describe our model. Section 3.1 will describe Flow and show how causal guarantees can be supported over independently replicated data. Section 3.2 describes an optimistic variation of Flow that provides linearizability in failure-free operation. Section 3.3 will extend our approach to providing generalized consensus guarantees and strong guarantees of linearizability. Finally, we conclude with relevant prior work funded by NSF support in Section 6, our timeline for our new work in Section 5, and discuss our broader impact in Section 7.

2 Background

This section describes the consistency guarantees we will support, together with intuition for how they relate to loosely-coupled wide-area systems.

Distributed object systems usually support one of three consistency levels: either best-effort [83, 40, 34], eventual, or causal [6]. Systems supporting just best effort [86, 84, 33] allow updates to be created at replicas without any serialization, *optimistically* assuming that the same objects will not be updated by multiple replicas at the same time (a "conflict"). A notification is made to high-level application layers or the user if a conflict is detected.

Eventually-consistent systems [103, 88, 30, 1, 5, 51] make the additional guarantee that the state of a single object eventually converges after updates cease. Concurrent updates of a single object might allow replicas to observe inconsistent state, but that state is guaranteed to converge at some undefined point in the future. Eventual consistency can be efficiently implemented without global serialization, requiring only a well-defined predicate to order conflicting updates. However, the application layer must be able to tolerate inconsistent views of the data.

Causally-consistent systems [6, 69, 70] respect potential causality defined by the *happens-before* ("hb") [56] relation. Access a_i happens before a_j if and only if:

1. a_i and a_j were both performed by a single replica, and a_i was performed first, or
2. a_i is a write whose data was returned by read a_j, or
3. the transitive closure of the above.

Causal consistency can also be implemented without additional messages (as with eventual consistency), but requires dependence information to accompany each update. For example, if y is updated after x is read ($x \overset{hb}{\rightarrow} y$), then the dependency on the x update must accompany the update to y. Additional message traffic is not required, though it does add metadata to existing messages. However, a replica receiving the update to y must satisfy the dependency on x before making the update to y visible, typically by passively waiting until the x update arrives. Causal consistency is the strongest consistency that can be efficiently implemented in an always-available system [72].

Stronger consistency levels, such as sequential consistency [57] or linearizability [41], can be supported at the potential cost of stalling during network partitions or with replica failures. To a first approximation, neither of these are supported by any current scalable wide-area distributed system except Spanner [23] (though systems built on chain-replication [104, 102, 7] can scale well under certain assumptions), which leverages a real-time infrastructure and strong availability assumptions to provide linearizability.

The advantage of stronger models is enormous simplification of the application or protocol designer's job. However, implementing sequential consistency requires all read responses to behave as if all accesses are totally ordered, as on a single-core uniprocessor. Linearizability additionally requires writes to conform to real-time ordering. If $t_{w_i} < t_{w_j}$, where t_{w_i} is the time w_i is performed in wall clock time, then w_i must appear before w_j in the corresponding total ordering. Said another way, both impose constraints on which values may be returned by reads, and these constraints are not resolved until write serialization is performed. Consider the canonical example:

R_1	R_2
w(y) 1	w(x) 1
r(x) 0	r(y) 0

Assuming both objects start with value zero, R_1 must conclude that its write to y must occur before R_2's write to x in any equivalent total ordering, whereas R_2 must conclude the reverse. This example is legal under causal consistency, but not under sequential consistency. A sequentially consistent system must delay a read until the serialization order of all writes before the read in the equivalent total order has been

determined. The reads would then return either $(1, 1)$, $(0, 1)$ or $(1, 0)$ for x and y. Determining serialization order can add enormous overheads in loosely-coupled systems.

Local versus non-local consistency models: Term a *local* consistency model one that can be entirely implemented by examining a local copy of data. By contrast, we term a *non-local* consistency model as one that requires active coordination between replicas to serialize accesses to data, such as supporting *coherence* on individual data objects, sequential consistency [56], or the even stronger linearizability [41]. Consider the above example again: replica R_1 writes to y and then attempts to read x. Under sequential consistency, a correct read can return 0 only if the write to x is ordered *after* the write to y. However, R_1 does not necessarily know that the write to x even exists, much less where it is ordered. Therefore, R_1 must delay the read of x until it can prove that there exists no such write anywhere in the system, potentially a lengthy delay for a system with varying connectivity.

The point of this discussion is that a passive replica can obtain the above assurance only by waiting until it has received information from every replica in the system, proving that none have created such a write. This passive approach is not guaranteed to make progress: a device that never communicates could halt the entire system. Additionally, a loosely coupled system with varying network connectivity could create lengthy delays.

Progress *can* be guaranteed if all writes are explicitly serialized through a single replica, and the replica remains available. We therefore say that sequential consistency and linearizability are *non-local* consistencies in that the system must actively serialize updates to achieve correctness (progress), as opposed to the weaker consistencies where passive approaches suffice. Stated another way, a replica can make a local decision about whether a view derived from local objects provides eventual or causal consistency. A replica cannot efficiently make a local decision about whether a local view is sequentially consistent; the system must provide global serialization mechanisms.

Related systems: Scatter uses nested consensus, based on a two-phase commit protocol across Paxos instances, to implement a key store [38]. Though all routing and key-mapping state is linearizable, Scatter actually only guarantees coherence to individual data items. Operations on different keys are not synchronized with respect to each other.

Megastore [12] supports ACID semantics across objects, but with two caveats. By Google's own admission [23], Megastore has poor write latency. More seriously, Megastore only supports strong consistency within a data partition, but not between data partitions.

3 Proposed work

The work described in this proposal assumes a loosely-coupled set of devices in the wide-area. We will assume that each device hosts a single replica. Clearly a device could host multiple distinct replicas, but they would have the same failure mode. Replicas communicate through standard network protocols, and are therefore subject to delays, partitions, and message losses. However, our failure model is fail-stop, we do not tolerate byzantine failures [60]. Though security is not explicitly discussed in the proposal, we are building systems that replicate data across users and devices, so authorization and access control will be part of the system, rather than the underlying devices.

The base set of desired functionality is a set of replicas that securely make shared data durable, and consistent, for dynamic and mobile groups of users. We approach this by designing a system, Flow, in which loosely-coupled replicas, which might live on a variety of differing devices and services, replicate amongst themselves through asynchronous gossiping [103, 54]. They annotate data with dependency information that allows each replica to locally export views with the relatively strong guarantee of causal consistency [6].

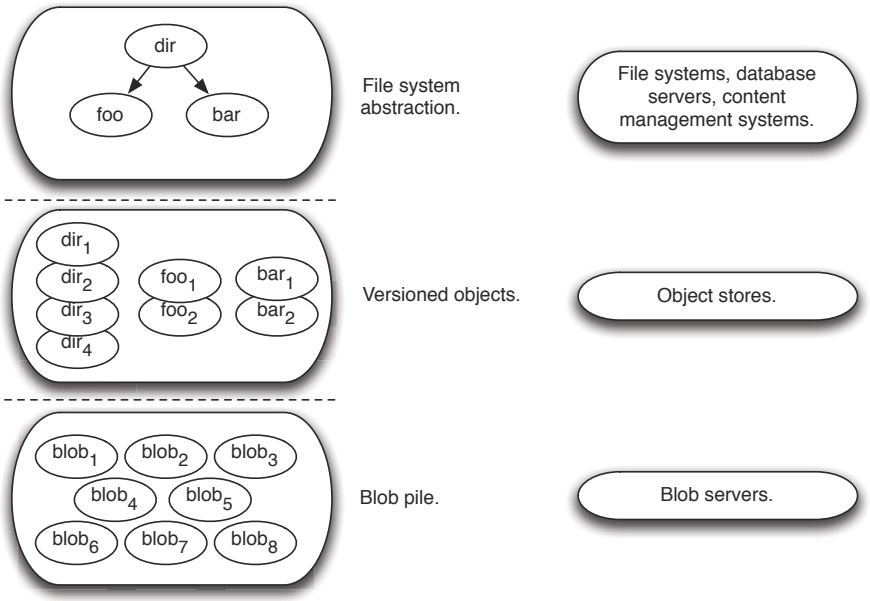

Figure 1: Data and servers in Flow. A blob pile, versioned objects, and a file system directory handled by blob servers, object stores, and a file system, respectively. Files and database tables are derived from versioned objects, which are derived from opaque, untyped blobs.

Researchers have demonstrated parts of this before. Our desired replica structure is similar to Camlistore [32], though Camlistore does not support consistency guarantees. DepSky [16] supports a durable data abstraction by layering on top of multiple cloud services. Bolt-on consistency [10] demonstrated the ability to support higher consistency levels on top of devices and services that export only lower consistency levels.

However, Flow has several attributes that will make it unique in the world of distributed systems. First, the system's intended environment is as discussed above: dynamic sets of devices for a single users or a small-scale collaborating group. This implies that the system must tolerate frequent disconnections, frequent membership changes, and a wide range of differing connectivities among connected devices. Second, the use of a local consistency model allows us to build the system with data replication separate from the consistency layer. This gives us the opportunity to build a robust, flexible system that can use a variety of data replication techniques, while supporting consistency guarantees without network communication.

Figure 1 shows the primary abstractions in Flow and the follow-on systems. Data can be viewed at three different levels: uninterpreted opaque blobs, versioned objects, and high level objects such as files or database tables. The three data layers are implemented by a software stack of blob servers, object stores, and applications, respectively. Figure 2 shows a rough diagram for the complete system. The system is based on blob servers, whose data is synthesized into versioned objects by object stores, which are then used by high-level applications such as file systems, databases, etc. The only communication between replicas is through blob servers, which replicate uninterpreted blobs among themselves on a best-effort basis. Blob servers and their blobs do not have to be co-resident. For example, both a blob server and its blobs could reside on a laptop, or a blob server could be on a mobile phone while the actual blobs are hosted on a cloud service. The same blobs can be accessed through multiple interfaces, such as a blob being exported through a file system, and through a query interface to a relational database.

3.1 A causally-consistent base of blob servers

The unit of replication is a set of blobs called a blob "pile". Blob piles should only be replicated by blob servers of authorized replicas. For example, assume a blob pile is defined by a unique tag and a shared

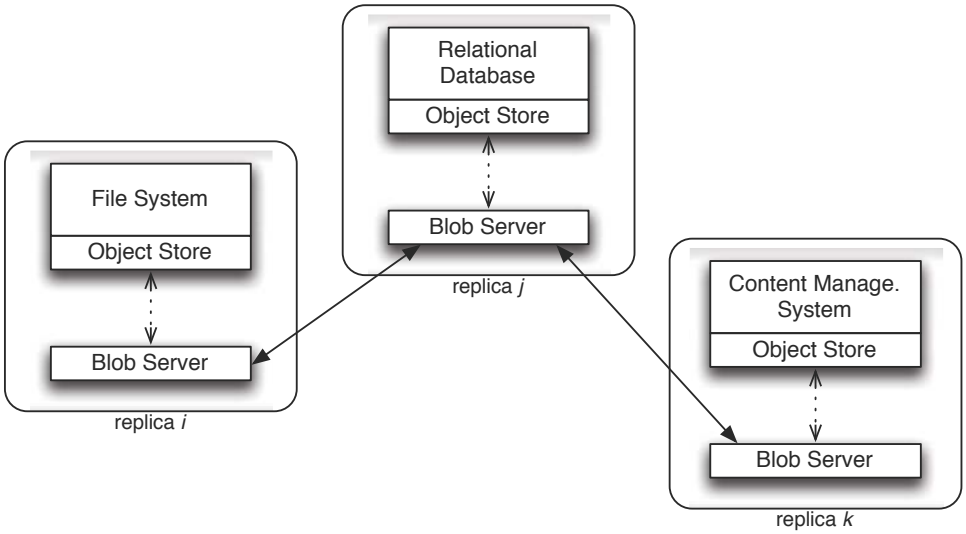

Figure 2: Flow: Untyped blobs are replicated by blob servers. Object store servers create local object stores by interpreting blobs that are replicated locally. Versioned objects from the object stores are interpreted as high-level applications such as file systems, database tables, or content management systems (CMS).

secret. Replicas could find each other through some combination of manual input, local area exploration and cryptographically secure introductions through other devices known to both parties [33, 28, 8]. Two replicas decide which piles to replicate by using public keys to establish a secure channel, and then exchanging pile tags. Each replica proves its right to replicate a pile by responding to a challenge with a MAC created from the challenge and the pile's secret.

A blob is an untyped, opaque chunk of data. Blobs are self-describing in that they are named by a combination of a hash algorithm name and the result of that hash being applied to the blob. Blobs could encapsulate data, byte ranges from objects or files, or meta-data, attributes of high-level objects. These latter *meta-blobs* are described as JSON [4] objects, which are collections of key/value pairs. Meta-blobs look no different than other blobs to the blob server. The intent is to make data self-describing to the extent that future data archaeologists will be able to decipher the data. JSON has the twin advantages of having unambiguous syntax, and in being human readable.

We here assume that blobs are not encrypted, as key management might be more profitably done by a higher software level. Blob servers are only required to be able to create, read, write, and list blobs. The blobs could live on local devices or cloud services. The blob servers could implement encryption, local replication, or erasure coding, as long as it is transparent to higher layers.

Blob servers and their replication of blobs is entirely distinct from object stores, in which reside higher-level properties such as consistency. Decoupling replication and consistency opens opportunities for different mechanisms and policies. For example, blob servers could replicate through two-way exchanges, one-way anti-entropies, or broadcasts through minimum-latency spanning trees, and do so without regard for ongoing transactions or data accesses. For purposes of this discussion, we will assume that blob servers replicate blobs through periodic, randomly-directed anti-entropy sessions [26, 103, 47, 15, 73].

A randomly-directed anti-entropy session consists of a single source replica pushing new updates to a randomly chosen target replica. Randomized anti-entropy allows the system to easily cope with dynamic situations that more rigid models would not handle. Our assumption so far is that blobs are immutable and permanent [90, 78, 42, 87, 73]. We defer discussion of deletion and garbage collection to Section 4, so an anti-entropy session here consists of determining which blobs are present at the source, but not the target. Good anti-entropy performance would require efficient summarization of a blob pile held by a blob

server. Blob sets could be summarized by version vectors (under strong assumptions on caching), Bloom filters [19] (potential false positives), or Merkle trees [76]. While both version vectors and Bloom filters have distinct advantages, only Merkle Trees provide complete and correct information.

Causal consistency: The following outlines a simple approach to supporting causal consistency. Each replica increments a local write count, wc_i, on every write. Write counts are used to maintain version vectors [74], that describe the number of writes a replica has performed from each replica in the system. For example, $vv_i = \langle 2, 4, 3 \rangle$ means that replica R_i has seen and performed 2 writes from R_0, 4 from R_1, and 3 from R_2. Version vector elements are incremented whenever writes are performed from either the local, or other replicas, and all writes from a given replica are applied in order (no holes). The local vector is also set to the pairwise max of its prior value and that of the starting vector for each write applied locally.

A replica modifies an object by creating a *write record* in the form of a new meta-blob. The write record includes edit operations defining the actual modification, the creating replica id R_i, the local write count, and a version vector that describes the local state prior to the write. Applying a write record at a replica means applying the modifications to the replica's object store. Write records are replicated as untyped blobs by the underlying blob server replication.

A replica receiving a write record cannot apply it until all other writes implied by the write's version vector have been applied. For example, let $w_{i,j}$ be the j'th write by replica R_i, and $w_{i,j}\langle x, y, z \rangle$ be the same write with the starting vector $\langle x, y, z \rangle$ specified. Assume R_0 has current vector $vv_0 = \langle 2, 2, 2 \rangle$, and receives new write $w_{1,3}\langle 2, 2, 4 \rangle$. The new write cannot be applied until R_0 has already received and applied missing writes $w_{2,3}$ and $w_{2,4}$. Since replication of blobs, and therefore new object writes, is independent of the consistency mechanism, R_0 does not request the writes directly. Instead, it merely waits until they arrive via the normal blob replication mechanism.

Concurrent writes by different replicas do not have to be applied at the same order by other replicas. For example, $w_{0,3}\langle 2, 2, 2 \rangle$ and $w_{1,3}\langle 2, 2, 2 \rangle$ are concurrent; each write was created before the other write had been applied locally. However, causal consistency only specifies a partial ordering. Therefore, correctness is not violated if $w_{0,3}$ is be applied before $w_{1,3}$ at R_3, but after $w_{1,3}$ at R_4.

Note that the loose structure of replicating blob servers impacts the representation of version vectors. We have been assuming that there is no well-known or static set of communicating replicas. The version vectors described above can therefore not be implemented as simple vectors indexed by replica IDs starting at 0. Instead, each replica chooses a random replica ID (RID) from a large space, and tags its writes with this RID. The version vector is then implemented as a set of tuples like: $\langle (RID_i, wc_i)...(RID_j, wc_j) \rangle$. We nonetheless continue the vector style in examples for clarity.

Convergent causal consistency: Causal consistency does not imply eventual consistency, but the two can be combined. Adding eventual consistency to causal consistency merely requires concurrent updates to the same object to applied in the same order everywhere. To the above mechanism, we can add the following: (i) each write is labeled with a create time, and (ii) each object is labeled with the create time of the last write applied, and (iii) an incoming write whose version vector shows that all dependencies have been is only applied if its create time is later than the object's last write time.

Ties between writes of different processors are broken through a well-known, deterministic procedure, such as comparing unique replica IDS. Synchronized clocks might make the sequence of intermediate object versions more palatable, but are not necessary for correctness. Object contents necessarily diverge, but the last write applied at each replica will always be the same.

Note that this approach only works when incoming writes completely overwrite an object. If writes instead are object *diffs*, or sequences of insertions and deletions of byte ranges at specific object offsets, the matter is more complicated.

Imagine write w_i extends a file by ten zero bytes, and w_j deletes the last five. If w_j is ordered before w_i,

the file should always end with at least ten zero bytes. However, if w_i is applied first, the wrong bytes will be deleted. This is a well-known problem, with a canonical solution of *operational transformations* [27, 31]. With operational transforms, w_i can still be applied immediately when it arrives. When w_j arrives and it is determined that it is ordered before the already-applied w_i, w_i must essentially be rolled back, and then w_j and w_i are applied in order.

Recovering from failure: Because Flow relies only on local information, the local view can be reconstructed from scratch by reading locally-replicated blobs from disk after a replica failure or reboot. On startup, the local object store re-parses local blobs and re-creates a set of versioned objects, each object complete with dependency data. The set of objects and their starting vector dependency data allows the object store to identify causally-consistent cuts [10, 74] of the objects. In particular, a restarting object store would attempt to find a *maximal cut*, i.e. the latest causally-consistent view of the objects.

As an example, consider the following set of writes found in a blob blob server by an object store after a reboot:

$$\begin{array}{ll} w_{1,1}\langle 0,0,0\rangle & \\ w_{1,2}\langle 0,1,0\rangle & \longrightarrow\ w_{1,2} \\ w_{2,1}\langle 0,0,0\rangle & \longrightarrow\ w_{2,1} \\ w_{2,2}\langle 1,0,1\rangle & \end{array}$$

As the listing shows, writes $w_{1,2}$ and $w_{2,1}$ compose a maximal causally-consistent cut of the four writes on the left. Write $w_{2,2}$ relies on $w_{0,1}$, which is not present.

Explicit dependencies: Causal relationships between blobs can be created in two ways. The discussion so far assumes potential causality, where a write is potentially causally dependent on every preceding object read. This can be overly conservative, and several researchers have instead assumed that causality would be explicitly specified [9, 10, 69]. Supporting explicit causality would not change the overall structure of this system, but it would change the details of dependency information. Instead of marking each write with a version vector, each write would explicitly list its antecedent write. Finding a causal cut, then, would consist of find the maximal set of writes, s.t. each member's antecedents is also in the set. Though this method of specifying dependencies is no slower when verifying a single write, it may be more difficult to find a maximal set.

A more dynamic approach: The communication model among blob servers has so far been described as completely randomly and egalitarian. However, our target scenarios provide opportunities to improve user-visible responsiveness and overall system efficiency at the same time. In particular, replicas might use local activity to inform the replication process. A replica being actively used by a user may try to quickly become up-to-date with respect to all other replicas. Inactive replicas might replicate at a low priority, "trickling" data in slowly to avoid affecting the bandwidth characteristics of higher priority processes [50, 93].

As an example, a user working on Flow data in the office might go home and work on the desktop in his den. The work replica transitions from active to inactive, while the den replica would react to the increased user activity by sending requests to other replicas for updated data. An active replica might scale replication activity back to avoid impacting the user's view of network behavior [105]. Upon transitioning to inactive status, however, the replica might push recent writes to other known replicas. Further, heuristics might be used to allow the replicas to make the transition pro-actively by using past transitions to anticipate the future. The office replica might actively try to update the home replica as soon as it transitions to inactive, resulting in the data being ready when the user arrives home. These approaches are similar to the notion of data flowing to where it is needed via introspection in the original OceanStore design [52].

3.2 Abstractions at the object store

In this section we describe a set of abstractions that allow communication to be optimized, stronger consistency guarantees to be embraced, and replicated blob piles to display different *aspects* on different replicas. However, Flow's loose structure is the key to fit in the mobile device environment. Rather than specifying restrictions on lower Flow layers, then, the work described here will merely inform lower layers, and be layered on top.

The previous section concluded with a discussion of informing the replication process through observations of the probable role of the local replica. This section carries this further, through definition and use of dynamic object *tablets* in object stores, the middle layer of our system.

We define a tablet as an enumeration of a set of semantically-related versioned objects. There are any number of ways we can define these sets using cluster analysis [53, 46], but a simple approach is to leverage our target use cases. Assume usage of distinct replicas is episodic. A given replica will see a quiescent period, followed by an active period, followed by another quiescent period. We can define the set of versioned objects accessed during this active period as a tablet. Tablets have several potential uses, including prefetching or replication prioritization, owner assignment, and sharing granularity definition. We discuss each in turn.

Replication prioritization: The replication strategy discussed so far is through randomized anti-entropy sessions. We could also use the related *rumor-mongering* [81, 29], in which replicas that have new information initiate anti-entropy "push" sessions. Those without new information are mostly inactive. Tablets come into play when deciding which replicas to ask for data, and deciding which data is appropriate.

The following is a pull-based approach. Assume active replicas update other replicas only slowly, to avoid swamping communication links that might be needed by application. Rather than sending data, they might analyse local behavior, use cluster analysis to define a dynamic tablet summarizing local accesses, and broadcast the tablet definition. Imagine a second replica receives an access request after a long fallow period. The request could be satisfied by a possibly-stale version in the local object store. However, if the requested object was part of a recently broadcast tablet, the replica could initiate a "pull" anti-entropy session with the other replica, specifying the desired tablet as a description of high-priority items.

This approach has several advantages. Continually keeping all replicas up to date could consume large amounts of bandwidth, and potentially overload resource- or energy-constrained mobile devices. Very low background replication traffic, and targeted foreground pulls, could limit overall traffic while ensure that active replicas are quickly brought up to date. Further, poor tablet definitions only affect performance, they do not violate correctness.

One potential issue is that the pulls defined above are at the semantic level, which consists of object tablets. Replication is done at the layer of blobs, not objects. However, local knowledge suffices for object stores to inform the blob server by enumerating the set of blobs corresponding to a set of object versions.

Optimistic linearizability: Stronger consistency models can be implemented by serializing accesses through an object *owner*. Assume each object o_x has a migrating owner, and that a replica must become an object's owner before accessing it. A replica reading o_x will then attempt to locate the owner through some combination of a default (perhaps a single distinguished replica, or the replica that created the object), soft state at other replicas, and path collapsing [68]. For example, assume o_x's default owner is R_j, which has since ceded ownership to R_k. Before accessing o_x, another replica R_i will become the owner by visiting R_j and then R_k. Both R_j and R_k will use soft state to record R_i as the new owner after responding to the requests.

Becoming an owner has consistency implications equivalent to an anti-entropy session from the old owner to the new owner, with the result of effectively stronger consistency than causal consistency being implemented. More formally, linearizability can be satisfied if we meet two restrictions [85]:

1. All accesses to each individual object x, including reads, must be totally ordered.
2. All reads must be "legal", where legal means that the read returns the most recent (in real time) value written to x.

These restrictions are both satisfied in the ownership scheme discussed above. Further, the assumption of episodic behavior means that the approach should be efficient, as ownership of an object should only change a single time when a user ceases using one device and moves to another.

Ownership migration: This discussion outlined a relatively simple migratory data ownership approach. We plan to investigate a variety of policies for determining when, and where, to migrate ownership in a variety of different single- and multi-user scenarios. The above approach could be termed migrate-on-access, and might be improved by introducing small amounts of hysteresis.

We will also investigate competitive migration protocols [92]. Both Black [18] and Karlin [45] studied variations of data migration across network topologies that satisfy the triangle inequality. Their approaches result in strongly 3-competitive online algorithms, algorithms that are provably within a factor of 3 of the best offline algorithm, and with the best possible worst case performance for an online algorithm. These results are not directly useful because they are centralized and rely on global knowledge. However, they will inform our own approaches, and do establish a baseline for good online performance.

Adaptive object granularity, and tablets: Assigning ownership on the granularity of individual objects might be too fine-grained for efficient performance. For example, if R_i accesses n distinct objects for a period of time, followed by R_j accessing the same n objects, n distinct ownership migration decisions have to be made. Each migration consumes time and resources. If the n objects were combined into one, the cost of the migration could be reduced.

We can instead assign ownership of complete tablets, either defined through cluster analyses, or through observations of similar or diverging access patterns among distinct objects. Each tablet will have a single owner, which need not be the same as the owner of any other tablet. Tablets can split and combine as needed *at runtime*. A tablet may be split if the runtime system detects that distinct portions of the tablet are being consistently accessed by different replicas. Multiple tablets might be combined if the runtime system detects that they are being accessed in similar ways.

Splitting and combining tablets is a relatively heavyweight operation, so we expect the best policies to include hysteresis, and to default to higher granularities. For example, one policy might be to start with a single tablet encompassing the entire object space, splitting only under duress, and favoring recombination unless there are continuing access patterns that favor the split.

Policies for adaptive granularity and for adaptive ownership are relatively orthogonal, though an actual system would make both tablet split and combination decisions at the owners. Adapting tablet granularities is just as prone to inconsistencies resulting from failures as is ownership migration. Unlike problems with ownership migration, however, these inconsistencies affect only performance; they do not cause violations of correctness.

These policies are similar in scope, if not in details, to protocols for dynamic cache line adjustment in shared memory multiprocessors [107, 49].

Object store aspects: An aspect can be thought of as a flattening of the set of objects exported by an object store in one or more dimensions, and is part of the local replica configuration. Let o_s be the set of objects contained in object store s. A *temporal flattening* of s is a slice of the object store's version history corresponding to the set of last updates prior to some time t, for each object in o_s. For example, a replica wishing to mount a snapshot of a file system at a time t would define the object store's local aspect to be a temporal flattening at time t.

A replica might wish to set the local aspect to show versioned objects, but with only the *landmark* versions visible, i.e., the last version in each cluster of per-file updates [90]. Allow o_{x,t_i} to denote the version of o_x at time t_i in s. We can then define a flattening through a predicate as follows: $\{o_{x,t_1} \mid \nexists o_{x,t_2} \wedge (t_2 - t_1) < \tau\}$, where τ specifies a minimum gap between clusters.

A resource-constrained mobile device might use a *spatial* flattening of s to make visible only a subset of the object store's object space. This subset is defined logically through a predicate across metadata tags [88, 82]. Tags are arbitrary ⟨*name,value*⟩ pairs that are defined in object metadata. For example, a replica on a phone might subscribe to an object store's aspect by specifying "`MP3 ∩ (bitrate < 128K)`".

A *consistency flattening* would allow a replica to specify a desired consistency level for the local object store aspect. Stronger consistency levels impose performance overheads, so replicas on resource-constrained or weakly-connected devices might opt for weaker semantics. The set of possible consistencies should clearly include eventual and causal, but might also include probabilistic guarantees [11].

Correctly and efficiently implementing flattened aspects might require an extra communication pathway from the object store down to the blob server. We have implied that one motivation for flattened aspects is to reduce a replica's storage requirements, as replicating only the most recent copies of versioned data objects would save space over replicating all versions, for example. However, as mentioned in our discussion of replication prioritization above, replication is done in the blob server layer, where blobs are not yet interpreted as versioned objects. Several approaches and policies might help. For example, a resource-constrained device might only replicate meta-blobs, which could be distinguishable by the blob server through format conventions of the JSON objects, and then fetch needed data blobs on demand.

A more promising approach is to define a portable and high-level *aspect expression*: something akin to an enhanced logical predicate or regular expression. A smartphone could pull only relevant data by presenting the aspect expression to each anti-entropy partner. The partner would parse the expression, find matching objects, and then enumerate local blobs that match.

Failures: Sadly, replicas failures break the linearizability guarantees potentially provided by the above scheme. Assume R_i becomes the owner of some object o_x, writes new values to o_x, and then fails. The first issue is that the new writes might not have been replicated elsewhere, causing write values to be lost. The above requirement of legal reads is then violated.

Worse, assume R_j fails before R_i requests ownership. R_i will not receive a response, and never find out that R_k is the current owner. R_i has two choices: it could either broadcast to the "group", or optimistically assume it is the new owner. The former approach relies on strong notions of group membership that are not supported by Flow-l structure. In the latter approach, multiple replicas might simultaneously believe themselves to be owners. However, the damage could be limited by checking for inconsistent ownership information, or concurrent updates to the same object, during anti-entropy sessions. Such a conflict would cause one of the replicas' claims to be owner to be invalidated.

This approach again does not support linearizability guarantees, but it could qualitatively improve user experiences.

3.3 Flow-p (generalized consensus)

Section 3.1 described an approach to loosely-coupled systems that can implement causal consistency in dynamic environments. Section 3.2 discussed approaches to using high-level semantic information to inform caching behavior, and to support linearizability during failure-free operation. This section discusses approaches to supporting strong consistency for such systems in the presence of failures. We start by describing generalized consensus, and then discuss a set of optimizations possible in our environment.

Generalized consensus algorithms such as Paxos [58] (alternatively Raft [79]) provide global safety (correctness) and liveness properties. Safety is always guaranteed, while the conditions where liveness is not guaranteed are difficult to provoke. Distributed consensus protocols are used by replicated state machines to implement a replicated log or, alternatively, establishing a single, durable ordering of new values. These values could be either new entries to append to the replicated log, consistency actions (writes), or modifications to the consensus group.

With reference to the latter, standard consensus protocols assume the participation of replicas from a well-known static group. Standard Paxos can make progress despite f replica failures, assuming the group contains at least $2f + 1$ members. Modification of the set of group members must be agreed upon by the consensus protocol, just as other decisions.

This means that Flow-p will be structured differently than Flow (or Flow-l). First, the replicas are no longer loosely coupled. Replicas must explicitly join and leave the replica group. Second, new values written to shared data are serialized by instances of the Paxos protocol. Every shared access potentially requires group coordination, which implies network communication.

Standard optimizations: Flow-p will assume the optimizations collectively known as Multi-Paxos [55, 21, 75]. In particular, the two phases of failure-free Paxos are used to (1) elect a leader, and (2) accept a new value. Assuming a long-lived leader allows a Paxos instance to require only a single round of communication during failure-free operation. Further, by granting this leader a lease [39], it can respond to read requests without any coordination at all. If this leader corresponds to a Flow-l data object "owner" under episodic assumptions (with leases), reads are entirely local operations. Each write, however, still requires group agreement.

Witnesses: Lamport [59] notes that while Paxos requires $2f + 1$ replicas to tolerate f failures, only $f + 1$ need maintain the full replicated log. Paraphrasing, the other f replicas could be "witnesses" [80]: replicas that only participate in decisions when replicas fail, or when group membership or leadership changes. Witnesses therefore do not need to maintain full system state, and are appropriate roles for local, but resource-constrained, devices.

Serialization-only Paxos: We can reduce bandwidth requirements even further by noting that consensus serves two distinct purposes: serializing the set of operations performed, and ensuring that writes are durable. However, witnesses alone are sufficient for the first issue. One might allege that knowing which writes committed in which order is meaningless without having the data itself, but this statement is rooted in the assumption of data center environments. A Paxos instance exporting a highly-available service in a data center must be able to reconfigure and serve any data quickly.

However, immediate availability of the latest writes might be less important in a local context. Assume we build a consensus instance using only a single full replica and $2f$ witnesses. Failure-free operation will proceed with a total of $f + 1$ participants, as above. However, if write data is not included in Paxos values, only the leader has up-to-date information.

This information becomes inaccessible if the leader now fails. However, the data is gone only temporarily if the failure is intermittent, i.e., temporary network partition or software failure followed by a reboot, as the leader will have all data stored on local stable storage.

If the failure is permanent, data is lost. However, the *identity* of the missing writes is maintained. These write identities could be bubbled up through the software stack and presented to the user. Differentiating between a one-paragraph addition to a blog and a command to launch a missile could allow a user to compensate for the lost data with minimal disruption. This situation is very different than if the identity of the lost writes was unknown. In this latter case, the user has no means to assess the importance of the lost writes, or to take compensating actions.

We can also define an asynchronous full replica. This type of replica will be a full participant in the consensus, but data transmission is removed from the critical path of the consensus, and is later sent asynchronously. This approach is an improvement on the above, as even permanent failures will only lose data if the failure happens after a write and before the asynchronous updates propagate the write to other replicas.

Tuning failure guarantees: A consensus instance with n replicas can tolerate $(n-1)/2$ failures. However, not all replicas need participate in the consensus protocol. Instead, we can define a consensus group large enough to tolerate the desired number of failures, and group any remaining replicas into a non-consensus group. Assuming writes are marked according to the serialization order defined by the consensus group, replicas in the non-consensus group can become aware of new writes through the standard anti-entropy process. Note, however, that a non-consensus replica must perform a write by contacting a consensus replica.

Different replicas might also be weighted differently. Standard paxos is a quorum protocol with equal weights for all participants. While suitable for the homogeneous, conditioned environment of a data center, this distribution does not match well with our target environment. Instead, progress may be enhanced by skewing quorum weights towards highly available and well-connected replicas [35, 36].

Aggregation and disconnected operation: Paxos efficiency could be increased if we could amortize the cost of multiple shared writes against a single Paxos instance (decision). If the application on top of Paxos outputs results only at some distant point in the future, there is no correctness-related need to use separate instances. Instead, the local replica can group together any number of writes into a single aggregate write, and send the result through the Paxos mechanism to be ordered against all other writes.

There are two caveats. First, writes can only be grouped if there are no intervening reads. Any read must return the most recent value written, meaning that the Paxos consensus leader needs to be contacted. Blithely using cached results violates correctness. The only exception is if the local replica is the current leader with an unexpired lease.

Second, the driving application must not communicate with any other entity between the grouped writes. If R_i groups together writes $w_{i,1}$ and $w_{i,2}$, and updates a web page between the two, a failure after the web page update, but before $w_{i,2}$, results in the web page reflecting uncommitted, and potentially unrecoverable, data, violating correctness.

We will investigate systems approaches to detecting and intercepting externalities [24, 101]. For example, the protocol might optimistically attempt to combine any run of consecutive writes. However, if the application is detected attempting to write to a network socket, i) the socket write is delayed, ii) the shared writes so far are combined and sent through the consensus process, and then iii) the socket write is allowed to complete.

Disconnected operation is in some ways similar, possible only if a replica is the leader with an unexpired lease. If the lease expires while the disconnection persists, correctness requires local accesses to cease.

3.3.1 Paxos performance

The overhead of the Gaios fast Paxos implementation instances is largely hidden by local disk latencies [20]. Their environment assumes includes local disk accesses take milliseconds, or tens of milliseconds, while local-area RPCs take tens or hundreds of microseconds. Our environment turns this on its head: wide-area or cloud RPC's can easily take tens or hundreds of milliseconds [2], while mobile devices could easily take an order of magnitude more, given weak connectivity and energy-saving strategies. This makes Paxos more expensive in our environment. The above optimizations will help, but Paxos will likely remain more expensive in our environment than in the data center.

However, disk latencies are largely untouched over the last decade, while network latencies continue to decrease. The network connectivity of mobile devices, in particular, has increased dramatically over the

last few years. The implication is that the latency tradeoff between wide-area network writes and local disk writes will likely soon mirror the current tradeoff between data center networks and local disk accesses.

4 Other topics

Transactions: Both write and "get transactions" [69], where reads on a set of keys return causally-consistent results, can be implemented locally in a blob pile by performing the set of accesses atomically with respect to new writes. Efficiently executing full read/write transactions is possible if the entire writeset has the same owner. Otherwise, a two-phase protocol will have to be layered on top, as in Walter [97].

Garbage Collection One of our central assumptions is that disk space is effectively free. This is true to a point, but there are still valid reasons to explicitly delete data from a blob pool. The central complication is ensuring deleted data do not return. We expect to use a standard approach, like deleting data by appending a distinguished version called a "tombstone".

Sharing We have not discussed ad hoc sharing here, though such sharing is central to our vision of how these systems will be used. However, flexible read-only sharing is easy to implement using capabilities, or even overloading `"http://"` for non-authenticated or password-authenticated data. More elaborate sharing can be implemented through lightweight replicas, similarly to resource-constrained support.

5 Project timeline

Our research plan spans three years. The budget contains funding for one RA the first year, and 1.5 RAs each of the last two years. The work described in this proposal directly affects the user experience of one of the most prominent portions of the operating system. Hence, we feel that incorporating a project-long user study is important for the design process. The following is our preliminary schedule:

- **Year 1:** (i) start longitudinal user study of Dropbox student use, (ii) flesh out abstractions and complete preliminary design, (iii) start Flow implementation, and (iv) design and implement file system and database services starting from existing systems built by our group.
- **Year 2:** (i) complete first full-on prototype of Flow and Flow-l, and (ii) move portion of user study onto Flow, (iii) develop consensus model and incorporate optimizations, and (iv) analyse expected and worst-case performance bounds for Flow-p.
- **Year 3:** (i) design and build Flow-p, and (ii) move portion of user study onto Flow-l, and (iii) start intensive benchmarking of files systems, transaction processing, and databases on top of system.

6 Results from Prior NSF Support

CSR—PDOS: Distributed Capability Systems (Keleher), CISE Grant 0720528, 8/1/2007 - 7/31/2012, $450,000.
This grant was motivated by the fact that traditional security models for wide-area file systems (passwords, Kerberos, ACLs), are slow, clumsy, and often provide access rights ill-suited to needs. These drawbacks might be corrected by building security around capabilities. However, traditional capabilities suffer from an inability to enforce confinement or accountability, and the inability to perform revocation at fine granularities.

Security features and restrictions, and identity models and requirements, are encapsulated in an object called a *chit*. We built a library implementation of chit functionality, and showed the generality of the chit mechanism by using it to build several applications using chit mechanisms to fully define and implement their security architectures. Our prototype chit library implementation provides a high-level interface for creating, deriving, verifying, and (un-)marshaling chits. Among others, our chit library [48] has been used to build Spore [13], a distributed system built from write-only storage, and T-Rex [66, 67] a system designed to minimise information leaks in consistency protocols.

CSR—PDOS: Data Staging and Parallel Applications in Robust Desktop Grids, Keleher PI, Richardson and Sussman co-PIs, CISE Grant 0916742, 9/1/2009 - 8/30/2013, $475,000.
This grant funded a broad investigation into decentralized bandwidth prediction [96, 94, 95], job scheduling on parallel machines[63, 64, 65], scheduling on multi-core machines [62, 61], and for parallel jobs [44].

7 Broader impact

Local Impact: Students trained during this project will learn a broad range of skills, including algorithmic design and analysis, mathematical modeling, systems building and measurement, and simulation. This will make them well-equipped to later attack other networking and security problems. We will also teach many of these ideas undergraduate classes, and use them as starting points for undergraduate research.

We expect to integrate both these ideas and resulting tools into our systems classes. At Maryland, we have a history of using research tools in our undergraduate curriculum [99, 98, 43]. Most of our undergraduate programming classes now use Marmoset [99], an automated submission and grading system. The current system is implemented on top of CVS; we intend to base the next version on top of Flow. This should give us a variety of real-world experience with which to fine-tune the system's interface before its release into the wild. During the final phase, we hope that project write-ups, supplied code, and submission will all be accomplished via Flow mount points, for multiple classes.

Non-academic: The most direct non-academic impact of this work will be a suite of open-sourced object stores that provide strong consistency guarantees, performance, and user control. We believe that, if successful, this project could have the broadest reach of any work that we have done. The target audience is not limited to large commercial enterprises that are loath to try new approaches. Instead, our target frankly is individual users and small groups who rely heavily on Dropbox [3], but find it fundamentally lacking in the attributes listed above. This is not to minimize the appeal, we are in this group as well. However, we feel that this is a rare opportunity to do thought-provoking research while reaching a large audience.

Outreach: Keleher plans to create a set of one-hour lessons on distributed systems issues, leading up to the work in this proposal, contrasted with popular schemes like Dropbox. Topics will include consistency definitions, consensus protocols, and byzantine protocols. The first target audience is the Montgomery Blair Math and Computer Science Magnet, with which the University of Maryland has a strong relationship. The end goal is to take this set of lectures to the feeder middle school magnet programs, where females and minorities have yet to drop out of the sciences in large numbers. Our hope is to help retention of females and minorities in the sciences, and in computer science in particular.

References

[1] Apache cassandra. http://cassandra.apache.org.

[2] Cloudping.info. https://www.cloudping.info.

[3] Dropbox. http://www.dropbox.com.

[4] JSON. https://www.json.org.

[5] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: Cap is only part of the story. *Computer*, 45(2):37–42, 2012.

[6] M. Ahamad, P. W. Hutto, and R. John. Implementing and programming causal distributed shared memory. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, May 1991.

[7] Sérgio Almeida, João Leitão, and Luís Rodrigues. Chainreaction: a causal+ consistent datastore based on chain replication. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 85–98. ACM, 2013.

[8] Inc. Apple Computer. Bonjour. http://developer.apple.com/networking/bonjour/.

[9] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 22. ACM, 2012.

[10] Peter Bailis, Ali Ghodsi, J Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *ACM SIGMOD*, 2013.

[11] Peter Bailis, Shivaram Venkataraman, Michael J Franklin, Joseph M Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment*, 5(8):776–787, 2012.

[12] Jason Baker, Chris Bond, James Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Léon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, volume 11, pages 223–234, 2011.

[13] Yunus Basagalar, Vassilios Lekakis, and Pete Keleher. Growing secure distributed systems from a spore. In *The International Conference on Distributed Computing Systems (ICDCS)*, June 2012.

[14] Nalini Moti Belaramani, Michael Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. PRACTI replication. In *NSDI*. USENIX, 2006.

[15] Nalini Moti Belaramani, Jiandan Zheng, Amol Nayate, Robert Soulé, Michael Dahlin, and Robert Grimm. PADS: A policy architecture for distributed storage systems. In Jennifer Rexford and Emin Gün Sirer, editors, *NSDI*, pages 59–74. USENIX Association, 2009.

[16] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. In *Proceedings of the sixth conference on Computer systems*, 2011.

[17] Ken Birman, Gregory Chockler, and Robbert van Renesse. Toward a cloud computing research agenda. *ACM SIGACT News*, 40(2):68–80, 2009.

[18] David L Black and Daniel Dominic Sleator. Competitive algorithms for replication and migration problems. Technical Report CMU-CS-89-201, Carnegie Mellon University, 1989.

[19] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the Association for Computing Machinery*, 13(7):422–426, 1970.

[20] William J Bolosky, Dexter Bradshaw, Randolph B Haagens, Norbert P Kusters, and Peng Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pages 11–11. USENIX Association, 2011.

[21] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 335–350. USENIX Association, 2006.

[22] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[23] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally-distributed database. In *Proceedings of OSDI*, volume 1, 2012.

[24] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174. San Francisco, 2008.

[25] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *SOSP*, volume 7, pages 205–220, 2007.

[26] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM, 1987.

[27] Clarence A Ellis and Simon J Gibbs. Concurrency control in groupware systems. *ACM SIGMOD Record*, 18(2):399–407, 1989.

[28] Carl Ellison and Steve Dohrmann. Public-key support for group collaboration. *ACM Transactions on Information and System Security (TISSEC)*, 6(4):547–565, 2003.

[29] Patrick T Eugster, Rachid Guerraoui, A-M Kermarrec, and Laurent Massoulié. Epidemic information dissemination in distributed systems. *Computer*, 37(5):60–67, 2004.

[30] A Feinberg. Project voldemort: Reliable distributed storage. In *Proceedings of the 10th IEEE International Conference on Data Engineering*, 2011.

[31] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. SPORC: Group collaboration using untrusted cloud resources. In Remzi H. Arpaci-Dusseau and Brad Chen, editors, *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 337–350. USENIX Association, 2010.

[32] Brad Fitzpatrick. camlistore. `http://camlistore.org`, 2014.

[33] Bryan Ford, Jacob Strauss, Chris Lesniewski-Laas, Sean Rhea, Frans Kaashoek, and Robert Morris. Persistent personal names for globally connected mobile devices. In *OSDI*, Seattle, Washington, November 2006.

[34] Armando Fox, Steven D Gribble, Yatin Chawathe, Eric A Brewer, and Paul Gauthier. Cluster-based scalable network services. In *ACM SIGOPS operating systems review*, volume 31, pages 78–91. ACM, 1997.

[35] Hector Garcia Molina, Frank Pittelli, and Susan Davidson. Applications of byzantine agreement in database systems. *ACM Transactions on Database Systems (TODS)*, 11(1):27–47, 1986.

[36] Vijay K Garg and John Bridgman. The weighted byzantine agreement problem. In *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 524–531. IEEE, 2011.

[37] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

[38] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable consistency in scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 15–28. ACM, 2011.

[39] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, 23(5):202–210, December 1989.

[40] Pat Helland. Life beyond distributed transactions: an apostate's opinion. In *CIDR*, pages 132–141, 2007.

[41] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[42] Dave Hitz, James Lau, and Michael Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, 1994.

[43] David Hovemeyer, Jeffrey K. Hollingsworth, and Bobby Bhattacharjee. Running on the bare metal with GeekOS. In *SIGCSE*, pages 315–319, 2004.

[44] Gary Jackson, Pete Keleher, and Alan Sussman. Parallel computing with P2P desktop grids. In *Submitted for publication*, December 2013.

[45] Anna R Karlin, Mark S Manasse, Larry Rudolph, and Daniel D Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):79–119, 1988.

[46] Leonard Kaufman and Peter J Rousseeuw. *Finding groups in data: an introduction to cluster analysis*, volume 344. Wiley. com, 2009.

[47] P. Keleher. Decentralized replicated-object protocols. In *Proc. of the 18th Annual ACM Symp. on Principles of Distributed Computing (PODC'99)*, May 1999.

[48] Pete Keleher, Bobby Bhattacharjee, Neil Spring Yunus Basagalar, and Vassilios Lekakis. Capabilities and identity in a wide-area file system. In *Submitted for publication*, June 2013.

[49] John H Kelm, Daniel R Johnson, William Tuohy, Steven S Lumetta, and Sanjay J Patel. Cohesion: a hybrid memory model for accelerators. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 429–440. ACM, 2010.

[50] Minkyong Kim, Landon P Cox, and Brian D Noble. Safety, visibility, and performance in a wide-area file system. In *FAST*, pages 131–144, 2002.

[51] J Kirkell. Consistency or bust: Breaking a riak cluster. In *OSCON '11*, 2011.

[52] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *ASPLOS*, 2000.

[53] Geoffrey H. Kuenning and Gerald J. Popek. Automated hoarding for mobile computers. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31,5 of *Operating Systems Review*, pages 264–275, New York, October 5–8 1997. ACM Press.

[54] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems (TOCS)*, 10(4):360–391, 1992.

[55] Lamport. Paxos made simple. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 32, 2001.

[56] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[57] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.

[58] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[59] Leslie Lamport and Mike Massa. Cheap paxos. In *Dependable Systems and Networks, 2004 International Conference on*, pages 307–314. IEEE, 2004.

[60] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[61] Jaehwan Lee, Pete Keleher, and Alan Sussman. Decentralized dynamic scheduling across heterogeneous multi-core desktop grids. In *19th International Heterogeneity in Computing Workshop*, April 2010.

[62] Jaehwan Lee, Pete Keleher, and Alan Sussman. Decentralized resource management for multi-core desktop grids. In *IEEE International Parallel and Distributed Processing Symposium*, April 2010.

[63] Jaehwan Lee, Pete Keleher, and Alan Sussman. Decentralized multi-attribute range search for resource discovery and load balancing. In *Accepted for publication in the Journal of Supercomputing*, 2014.

[64] Jaehwan Lee, Pete Keleher, and Alan Sussman. Exploiting multi-core nodes in peer-to-peer grids. In *Accepted for publication in JPDC*, 2014.

[65] Jaehwan Lee, Peter J. Keleher, and Alan Sussman. Supporting computing element heterogeneity in P2P grids. In *IEEE Cluster 2011 Conference*, September 2011.

[66] Vassilios Lekakis, Yunus Basagalar, and Pete Keleher. Don't trust your roommate, or, access control and replication protocols in home environments. In *The 4th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage*, June 2012.

[67] Vassilios Lekakis, Yunus Basagalar, and Pete Keleher. Sharing and information leakage in replicated storage systems. In *Submitted for publication*, June 2013.

[68] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.

[69] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, 2011.

[70] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13), Lombard, IL*, 2013.

[71] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. Consistency, availability, and convergence. *University of Texas at Austin TR-11-22 (May)*, 2011.

[72] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. Consistency, availability, and convergence. *University of Texas at Austin TR-11-22 (May)*, 2011.

[73] Ali José Mashtizadeh, Andrea Bittau, Yifeng Frank Huang, and David Mazières. Replication, history, and grafting in the ori file system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 151–166. ACM, 2013.

[74] Friedemann Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989.

[75] David Mazieres. Paxos made practical, 2007.

[76] Merkle. A digital signature based on a conventional encryption function. In *CRYPTO: Proceedings of Crypto*, 1987.

[77] Lily B Mummert, Maria R Ebling, and Mahadev Satyanarayanan. *Exploiting weak connectivity for mobile file access*, volume 29. ACM, 1995.

[78] Kiran-Kumar Muniswamy-Reddy, Charles P Wright, Andrew Himmer, and Erez Zadok. A versatile and user-oriented versioning file system. In *FAST*, volume 4, pages 115–128, 2004.

[79] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. `http://ramcloud.stanford.edu/raft.pdf`.

[80] J-F Pâris and Darrell DE Long. Voting with regenerable volatile witnesses. In *Data Engineering, 1991. Proceedings. Seventh International Conference on*, pages 112–119. IEEE, 1991.

[81] Boris Pittel. On spreading a rumor. *SIAM Journal on Applied Mathematics*, 47(1):213–223, 1987.

[82] Ansley Post, Juan Navarro, Petr Kuznetsov, and Peter Druschel. Autonomous storage management for personal devices with podbase. In *USENIX*. USENIX, 2011.

[83] Dan Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, 2008.

[84] Venugopalan Ramasubramanian, Thomas L. Rodeheffer, Douglas B. Terry, Meg Walraed-Sullivan, Ted Wobber, Catherine C. Marshall, and Amin Vahdat. Cimbiosys: A platform for content-based partial replication. In Jennifer Rexford and Emin Gün Sirer, editors, *NSDI*, pages 261–276. USENIX Association, 2009.

[85] Michel Raynal. Sequential consistency as lazy linearizability. In *EurAsia-ICT 2002: Information and Communication Technology*, pages 866–873. Springer, 2002.

[86] Peter Reiher, John Heidemann, David Ratner, Greg Skinner, and Gerald Popek. Resolving file conflicts in the ficus file system. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, 1994.

[87] W. D. Roome. 3DFS: A time-oriented file server. In *Proceedings of the Usenix Winter 1992 Technical Conference*, pages 405–418, Berkeley, CA, USA, January 1991. Usenix Association.

[88] Brandon Salmon, Steven W. Schlosser, Lorrie Faith Cranor, and Gregory R. Ganger. Perspective: semantic data management for the home. In *Proccedings of the 7th conference on File and storage technologies*, pages 167–182, Berkeley, CA, USA, 2009. USENIX Association.

[89] Jerome H Saltzer, David P Reed, and David D Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, 1984.

[90] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, pages 110–123, December 1999.

[91] Eric Schmidt and Hal Varian. Google: ten golden rules. *Newsweek*, 2005.

[92] Daniel D Sleator and Robert E Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[93] Sumeet Sobti, Nitin Garg, Fengzhou Zheng, Junwen Lai, Yilei Shao, Chi Zhang, Elisha Ziskind, Arvind Krishnamurthy, and Randolph Y. Wang. Segank: A distributed mobile storage system. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, 2004.

[94] Sukhyun Song, Peter J. Keleher, Bobby Bhattacharjee, and Alan Sussman. Decentralized network bandwidth prediction. In *24th International Symposium on Distributed Computing*, September 2010.

[95] Sukhyun Song, Peter J. Keleher, Bobby Bhattacharjee, and Alan Sussman. Decentralized, accurate, and low-cost network bandwidth prediction. In *The 30th IEEE International Conference on Computer Communications (IEEE INFOCOM 2011)*, April 2011.

[96] Sukhyun Song, Peter J. Keleher, and Alan Sussman. Searching for bandwidth-constrained clusters. In *The 31st International Conference on Distributed Computing Systems (ICDCS 2011)*, April 2011.

[97] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In Ted Wobber and Peter Druschel, editors, *Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011*, pages 385–400. ACM, 2011.

[98] Jaime Spacco, David Hovemeyer, and William Pugh. An eclipse-based course project snapshot and submission system. In *3rd Eclipse Technology Exchange Workshop (eTX)*, Vancouver, BC, October 24, 2004.

[99] Jaime Spacco, Jaymie Strecker, David Hovemeyer, and William Pugh. Software repository mining with Marmoset: An automated programming project snapshot and testing system. In *Proceedings of the Mining Software Repositories Workshop (MSR 2005)*, St. Louis, Missouri, USA, May 2005.

[100] Michael Stonebraker. Why enterprises are uninterested in NoSQL. `http://cacm.acm.org/blogs/blog-cacm/99512-why-enterprises-are-uninterested-in-nosql/fulltext`, 2010.

[101] Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Membrane: Operating system support for restartable file systems. *Trans. Storage*, 6(3):1–30, 2010.

[102] Jeff Terrace and Michael J Freedman. Object storage on craq: High-throughput chain replication for read-mostly workloads. In *Proc. USENIX Annual Technical Conference*, page 59, 2009.

[103] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.*, 1995.

[104] Robbert van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, pages 91–104, 2004.

[105] Arun Venkataramani, Ravi Kokku, and Mike Dahlin. Tcp nice: A mechanism for background transfers. *ACM SIGOPS Operating Systems Review*, 36(SI):329–343, 2002.

[106] Yupu Zhang, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. *-Box: Towards Reliability and Consistency in Dropbox-like File Synchronization Services. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '13)*, San Jose, California, June 2013.

[107] Hongzhou Zhao, Arrvindh Shriraman, Snehasish Kumar, and Sandhya Dwarkadas. Protozoa: Adaptive granularity cache coherence. In *In proceedings of the International Symposium of Computer Architecture*, 2013.

Biographical Sketch Dr. Peter J. Keleher

Department of Computer Science, The University of Maryland, College Park
email: keleher@cs.umd.edu, tel: +1-301-405-0345 January 14, 2014

(a) Professional Preparation

Rice University, Houston, TX; Electrical Engineering; BS, 1986
Rice University, Houston, TX; Computer Science; MS, 1993
Rice University, Houston, TX; Computer Science; Ph.D., 1995

(b) Appointments

2001-present: Associate Professor, University of Maryland, College Park, MD
1995-2001: Assistant Professor, University of Maryland, College Park, MD

(c) Products

- "Capabilities and Identity in a Wide-Area File System", Pete Keleher and Bobby Bhattacharjee and Neil Spring Yunus Basagalar and Vassilios Lekakis. Submitted for publication.
- "Growing Distributed Systems From a Spore", Yunus Basagalar, Vassilios Lekakis and Pete Keleher. In *International Conference on Distributed Systems*, June 2012.
- "File System Support for Collaboration in the Wide Area", Vasile Gaburici, Pete Keleher and Bobby Bhattacharjee. In *The 26th International Conference on Distributed Computing Systems*, July 2006.
- "Tapeworm: High-Level Abstractions of Shared Accesses", Pete Keleher. In *The 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.
- "Don't Trust Your Roommate, or, Access Control and Replication Protocols in Home Environments", Vassilios Lekakis, Yunus Basagalar, and Pete Keleher. In *The 4th USENIX Workshop on Hot Topics in Storage and File Systems*, June 2012.
- "Distributed Ranked Search", Vijay Gopalakrishnan, Ruggero Morselli, Bobby Bhattacharjee, Pete Keleher, and Aravind Srinivasan. In *14th Annual IEEE International Conference on High Performance Computing*, 2007. **Best Paper Award**.
- "Resource Discovery Techniques in Distributed Desktop Grid Environments", Jik-Soo Kim, Beomseok Nam, Peter Keleher, Michael Marsh, Bobby Bhattacharjee and Alan Sussman. In *The 7th IEEE/ACM International Conference on Grid Computing*, September 2006. **Best Paper Award**.
- "File System Support for Collaboration in the Wide Area", Vasile Gaburici, Pete Keleher, and Bobby Bhattacharjee. In *The 26th International Conference on Distributed Computing Systems*, July 2006.
- "Tapeworm: High-Level Abstractions of Shared Accesses", Pete Keleher. In *The 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, February 1999.

- "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems", Pete Keleher, Alan L. Cox, Sandhya Dwarkadas and Willy Zwaenepoel. In *The 1994 Winter USENIX Conference*, 1994.

(e) Collaborators & Other Affiliations

Collaborators and Co-Editors: none.

Graduate Advisers and Postdoctoral Sponsors: *Rice University:* Willy Zwaenepoel; Alan Cox.

Graduated Ph.D students: Sukhyun Song, Jaehwan Lee, Jik-Soo Kim, Ugur Cetintemel, Bujor Silaghi, Kritchal Thitikamol.