Flexible Consistency for Wide area Peer Replication

Sai Susarla and John Carter
{*sai, retrac*}*@cs.utah.edu*

UUCS-04-016

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

November 18, 2004

Abstract

The lack of a flexible consistency management solution hinders P2P implementation of applications involving updates, such as read-write file sharing, directory services, online auctions and wide area collaboration. Managing mutable shared data in a P2P setting requires a consistency solution that can operate efficiently over variable-quality failure-prone networks, support pervasive replication for scaling, and give peers autonomy to tune consistency to their sharing needs and resource constraints. Existing solutions lack one or more of these features.

In this paper, we describe a new consistency model for P2P sharing of mutable data called *composable consistency*, and outline its implementation in a wide area middleware file service called *Swarm*[1]. Composable consistency lets applications compose consistency semantics appropriate for their sharing needs by combining a small set of primitive options. Swarm implements these options efficiently to support scalable, pervasive, failure-resilient, wide-area replication behind a simple yet flexible interface. We present two applications to demonstrate the expressive power and effectiveness of composable consistency: a wide area file system that outperforms Coda in providing close-to-open consistency over WANs, and a replicated BerkeleyDB database that reaps order-of-magnitude performance gains by relaxing consistency for queries and updates.

[1] *Swarm* stands for Scalable Wide Area Replication Middleware.

1 Introduction

Organizing wide area applications in a peer-to-peer (P2P) fashion can improve availability, failure resilience, and scalability compared to traditional client-server architectures. Many effective P2P techniques have been developed to locate and share read-only content [14] or low-write-sharing content [3, 11, 15]. Although P2P organization could benefit wide area applications with mutable data such as online auctions, directory services, and collaborative applications (e.g., Lotus Notes), their data characteristics and consistency needs are diverse and inadequately supported by existing P2P middleware systems.

P2P sharing of mutable data raises the issues of replication and consistency management. P2P systems pose unique challenges to replication algorithms. P2P clients tend to experience diverse network characteristics and vary in their resource availability and willingness to handle load from other peers. The collection of clients tends to be large, but constantly changing. A P2P consistency management system must therefore be able to support pervasive replication to scale with load, enable peers to individually balance consistency and availability against performance and resource usage, and operate efficiently across non-uniform failure-prone networks.

To understand the diversity of application characteristics and their consistency needs, we surveyed a variety of wide-area data sharing applications ranging from personal file access (with little data sharing) to widespread real-time collaboration (e.g., chat and games, with fine-grain synchronization) [17]. The survey identified three broad classes of distributed applications: (1) file access, (2) database and directory services, and (3) real-time collaborative groupware.

We found that applications differ widely in the frequency and extent of read and write sharing among replicas, the typical replication factor, their tolerance to stale data, semantic interdependencies among updates, the likelihood of conflicts among concurrent updates, and their amenability to automatic conflict resolution. Some applications need different consistency semantics for reads and writes, e.g., an online auction service might need strong consistency for writes to prevent selling the same item twice, but could tolerate providing stale data in response to queries to improve performance. Applications differ in the degree to which they tolerate replica divergence, e.g., users working on a shared document expect it to reflect the latest updates made by other users before each editing session (*close-to-open* consistency), while the staleness of stock-quote updates should be bounded by a time interval. Applications access files in different ways with different consistency requirements, e.g., personal files are rarely shared while software and multimedia file are widely read-shared. Other applications frequently write share data, e.g., distributed logs, shared calendars, and version control systems. How concurrent updates can be handled

varies widely, e.g., it is relatively easy to combine concurrent appends (logging) or mergeable writes (calendars), but more complex sharing and write conflict patterns often require explicit serialization. For some applications, optimistic or eventual consistency (i.e., propagating updates lazily) provides adequate semantics and high availability, but during periods of close collaboration users need tighter synchronization guarantees such as close-to-open (to view latest updates) or strong consistency (to prevent update conflicts).

Based on the results of our application study, we believe that managing mutable shared data in a P2P setting requires a consistency solution that can operate efficiently over variable-quality failure-prone networks, support pervasive replication for scaling, and give peers autonomy to tune consistency to their sharing needs and resource constraints. Existing solutions fail to provide one or more of these requirements. Several P2P systems support file and database replication with updates by providing close-to-open or eventual consistency [11, 3, 15, 13], which is inadequate for many applications. A number of previous efforts [18, 7] have viewed consistency semantics as a continuous linear spectrum ranging from strong consistency to best-effort eventual consistency, and provided ways to control divergence of replica contents. While powerful, these systems need to be extended in several ways to cover the full spectrum of observed application needs.

When we carefully examined the results of our survey, we observed that the consistency needs of applications can be expressed in terms of a small number of design options, which we classify into five mostly orthogonal dimensions:

- *concurrency* - the degree to which conflicting (read/write) accesses can be tolerated,
- *replica synchronization* - the degree to which replica divergence can be tolerated, including the types of inter-dependencies among updates that must be preserved when synchronizing replicas,
- *failure handling* - how data access should be handled when some replicas are unreachable or have poor connectivity,
- *update visibility* - the time at which local modifications to replicated data become 'stable' and ready to be made visible globally, and
- *view isolation* - the time at which remote updates must be made visible locally.

There are multiple reasonable options along each of these dimensions that existing systems employ separately. Based on this classification, we developed a novel *composable consistency model* that provides the options listed in Table 1. When these options are combined in different ways, they yield a rich collection of consistency semantics for shared data that

Dimension	Available Consistency Options
Concurrency Control	**Access mode**: *concurrent* (RD, WR) | excl (RDLK, WRLK)
Replica Synchronization	**Timeliness**: manual | *time* (staleness = *0*..∞ secs) | *mod* (unseen writes = 0..∞) **Strength**: *hard* | soft **Semantic Deps.**: - | causal **Semantic Deps.**: - | atomic **Update ordering**: none | *total* | serial
Failure Handling	*optimistic* (ignore distant replicas w/ RTT > 0..∞) | pessimistic
Update Visibility	*session* | per-access | manual
View Isolation	*session* | per-access | manual

Table 1: Consistency options provided by the Composable Consistency (CC) Model. Consistency semantics are expressed for an access session by choosing one of the alternative options in each row, which are mutually exclusive. Options in italics indicate reasonable defaults that provide close-to-open semantics suitable for many applications. In our discussion, when we leave an option unspecified, we assume its default value.

cover the needs of a broad variety of applications. Using composable consistency, a P2P auction service could employ strong consistency for updates, while relaxing consistency for queries to limit synchronization cost. An auction user can specify stronger consistency requirements to ensure 100% accurate query results, although doing so increases overhead and latency.

In this paper, we describe the composable consistency model and outline how we implemented it in a wide area P2P middleware file service called *Swarm*. Swarm lets applications express consistency requirements as a vector of options along these dimensions on a per-access basis rather than only supporting a few packaged combinations. Doing so gives applications more flexibility in balancing consistency, availability, and performance. In addition, Swarm performs aggressive peer replication and tunes its replication behavior in response to observed network and node availability conditions. As a result, Swarm more effectively caches data near where it is accessed, thereby providing low latency data access, under a broader range of data sharing scenarios than existing systems.

To determine the value of composable consistency in Swarm, we implemented both a wide area P2P file system and a wrapper library that supports replication around the BerkeleyDB database library. We show that Swarm's composable consistency ensures close-to-open semantics over a WAN and outperforms Coda's client-server implementation by exploiting 'nearby' replicas during roaming file access. We also show that replicating BerkeleyDB us-

ing Swarm enables order-of-magnitude improvements in write performance and scalability beyond master-slave replication by relaxing consistency in several ways. Swarm relieves application writers of the burden of implementing their own replication and consistency mechanisms. Since the focus of this paper is on evaluating composable consistency in a wide area P2P replication environment, we limit our discussion of Swarm's design to its consistency implementation. A more complete description and evaluation of Swarm's design, including its dynamic hierarchical replication and its scalability and failure resilience properties, appears elsewhere [17].

In Section 2 we describe prior work on flexible consistency management and wide area replication management. In Section 3 we describe the composable consistency model. We briefly outline Swarm's design in Section 4 before describing its consistency implementation in Section 5. We present our evaluation of its practicality in Section 6. Finally, in Section 7 we conclude.

2 Related Work

Several solutions exist to manage replication among wide-area peers. Several P2P systems were developed to share read-only data such as multimedia files (PAST [14], KaZaa) or access to rarely write-shared personal files [15], but not frequent write-sharing. Numerous consistency schemes have been developed individually to handle the data coherence needs of specific services such as file systems, directory services [10], databases and persistent object systems [9]. Distributed file systems such as NFS, Pangaea, Sprite, AFS, Coda and Ficus target traditional file access with low write-sharing among multiple users. Ivy [11] is a read/write P2P file system that supports close-to-open semantics based on P2P block storage and logs. Composable consistency adopts a novel approach to support many of their consistency semantics efficiently in a P2P setting.

Bayou [3] explored optimistic replication and update propagation for collaborative applications under ad-hoc connectivity. Our consistency implementation employs some of their techniques in a wider context.

Fluid replication [2] provides multiple selectable consistency policies for static hierarchical caching. In contrast, our approach offers primitive options that can be combined to yield a variety of policies, offering more customizability for peer-to-peer replication. Many design options [8, 7] that we identified for composable consistency are proposed separately by prior systems. Our model forges them uniquely for flexibility.

Many previous efforts have explored a continuous consistency model [18, 7]. Of those, the TACT toolkit [18] provides continuously tunable consistency along three dimensions similar to those covered by our timeliness and concurrency control aspects. We provide additional flexibility including the notion of sessions and explicit semantic dependencies to cater to a wider variety of application needs as described in Section 3. TACT's order error offers continuous control over the number of update conflicts. In contrast, our concurrency options provide a binary choice between zero and unlimited number of conflicts. We believe that for many real-world applications, a binary choice such as ours is adequate and reduces bookkeeping overhead.

3 Composable Consistency Model

The composable consistency (CC) model is applicable to systems that employ the *data-shipping* paradigm. Such applications are structured as multiple distributed components, each of which holds a portion of application state and operates on other portions by locally caching them as needed. The CC model assumes that applications access (i.e., read or write) their data in sessions, and that consistency can be enforced at session boundaries or before and after each read or write access.

In the CC model, an application expresses its consistency requirements for each session as a vector of consistency options , as listed in Table 1. Each row of the table indicates several mutually exclusive options available to control the aspect of consistency indicated in its first column. Reasonable default options are noted in italics, which together enforce the close-to-open consistency semantics provided by AFS [5] for coherent read-write file sharing. A particular CC implementation can provide reasonable defaults, but allow an application to override the defaults if needed. Table 2 lists the CC options for several consistency flavors.

Concurrency: CC provides two flavors of access modes to control the parallelism among reads and writes. Concurrent modes (RD, WR) allow arbitrary interleaving of accesses across replicas, similar to Unix file semantics. Exclusive modes (RDLK, WRLK) provide traditional concurrent-read-exclusive-write semantics [12]. When both flavors are employed on the same data simultaneously, RD mode sessions coexist with all other sessions, i.e., RD operations can happen in parallel with exclusive sessions, while WR mode sessions are serialized with respect to exclusive sessions, i.e., they occur before the RDLK/WRLK session begins or are deferred until it ends.

Timeliness Guarantees: The degree to which replica contents are allowed to diverge (i.e., *timeliness* in Table 1) can be specified in terms of time, the number of missed updates,

or both. These options are analogous to the TACT toolkit's staleness and numeric error metrics [18]. The timeliness bounds can be hard, i.e., strictly enforced by stalling writes if necessary (like TACT), or soft, i.e., enforced in a best-effort manner for high availability. Two types of semantic dependencies can be directly expressed among multiple writes (to the same or different data items), namely, *causality* and *atomicity*. These options together with the session abstraction can be used to support the transactional model of computing.

Update Ordering Constraints: When updates are issued independently at multiple replicas (e.g., by 'WR' mode sessions), our model allows them to be applied (1) with no particular constraint on their ordering at various replicas (called '*none*'), (2) in some arbitrary but common order everywhere (called '*total*'), or (3) sequentially via serialization (called '*serial*'). The ordering options can be specified on a per-session basis. Unordered updates provides high performance when updates are commutative, e.g., increments to a shared counter or updates to different entries in a shared directory. Totally ordered updates can be concurrent, but may need to be reapplied to ensure global ordering. Some updates that require total order cannot be undone and reapplied, and thus must be globally serialized. For example, when multiple clients concurrently issue the dequeue operation at different replicas of a shared queue, they must not obtain the same item, although multiple items can be enqueued concurrently and reordered later. Hence the dequeue operation requires 'serial' ordering, while enqueue requires 'total' ordering.

Failure Handling: When all replicas are not equally well-connected, different consistency options can be imposed on different subsets of replicas based on their relative connectivity by specifying a cut-off network quality. Consistency semantics are guaranteed at a replica only relative to those replicas reachable via higher quality links. For instance, this enables a directory service to provide strong consistency among clients within a campus, while enforcing optimistic eventual consistency across campuses for higher availability.

Visibility and Isolation: Finally, a session can determine how long it stays isolated from the updates of remote sessions, as well as when its own updates are ready to be made visible to remote sessions. A session can choose to remain isolated entirely from remote updates ('session', ensuring a snapshot view of data), allow them to be applied on local copies immediately ('per-access', useful for log monitoring), or when explicitly requested ('manual'). Similarly, a session's updates can be propagated as soon as they are issued (useful for chat), when the session ends (useful for file updates), or when explicitly requested ('manual').

Although CC's options are largely orthogonal, they are not completely independent. For example, the exclusive access modes imply session-grain visibility and isolation, a hard most-current timeliness guarantee, and serial update ordering.

At first glance, providing a large number of options rather than a small set of hardwired protocols might appear to impose an extra burden on application programmers. However, thanks to the orthogonality and composability of CC's options, their semantics are roughly additive and application programmers need not consider all combinations of options. Rather, we anticipate that implementations of CC will bundle popular combinations of options as defaults (e.g., 'Unix file semantics', 'CODA semantics', or 'best effort streaming'), while allowing individual applications to refine their consistency mechanisms when required. In those circumstances, programmers can choose individual options along interesting dimensions while retaining the other options from the default protocol.

4 Swarm

Swarm is a distributed file store organized as a collection of per servers (called *Swarm servers*) that provide coherent wide area file access at a variable granularity. Applications store their shared state in Swarm files and operate on their state via nearby Swarm servers.

Files in Swarm are persistent variable-length flat byte arrays named by globally unique 128-bit numbers called SWIDs. Swarm exports a file system-like session-oriented interface that supports traditional read/write operations on file blocks as well as operational updates on files (explained below). A file block (also called a *page*, default 4KB) is the smallest unit of data sharing and consistency in Swarm. Swarm servers locate files by their SWIDs, cache them as a side-effect of local access, and maintain consistency according to the per-file consistency options (as described in Section 3). Each Swarm server uses a portion of its local persistent store for permanent ('home') copies of some files and uses the remaining space to cache remotely homed files. Swarm servers discover each other as a side-effect of locating files by their SWIDs. Each Swarm server monitors the connection quality (latency, bandwidth, connectivity) to other Swarm servers with which it has communicated in the recent past, and uses this information to form an efficient dynamic hierarchical overlay network of replicas of each file.

Swarm exports the traditional session-oriented file interface to its client applications via a Swarm client library linked into each application process. The interface allows applications to create and destroy files, open a file session with specified consistency options, read and write file blocks, and close a session. A session is Swarm's unit of concurrency control and isolation. A Swarm server also exports a native file system interface to Swarm files via the local operating system's VFS layer, ala CodaFS [6]. The wrapper provides a hierarchical file name space by implementing directories within Swarm files.

Swarm allows files to be updated in two ways: (i) by directly overwriting the previous contents on a per file block basis (called *absolute* or *physical updates*) or (ii) by registering a semantic update procedure (e.g., "add(name) to directory") and then invoking the `update()` interface to get Swarm to perform the operation on each replica (called *operational updates*). Before invoking an operational update, the application first must link a plugin to each Swarm server running an application component. The plugin is used both to apply update procedures and to perform application-specific conflict resolution. When operational updates are used, Swarm replicates entire files as a single consistency unit.

Using Swarm: To use Swarm, applications link to a Swarm client library that invokes operations on a nearby Swarm server in response to client operations (e.g., creating a file, opening an access session, or performing a read or write). When applications open a file, they can specify a set of consistency options. As described below, the local Swarm server interacts with other Swarm servers to acquire and maintain a locally cached copy of the file with the appropriate consistency. Figure 1 illustrates a database service and a directory service (both derived from the BerkeleyDB embedded database library [16]) using Swarm to implement wide area proxy caching. The 'DB' represents the unmodified BerkeleyDB database service oblivious to replication, the 'AS' represents the auction or directory service-specific logic, and the 'FS' provides local storage. Our evaluation in Section 6 shows the benefit of such replication.

Replication: The Swarm servers caching a particular file dynamically organize themselves into an overlay *replica hierarchy*. All communication between replicas happens via this hierarchy. Figure 2 shows a Swarm network of six servers with hierarchies for two files. One or more servers act as *custodians*, which maintain permanent copies of the file. Having multiple custodians enables fault-tolerant file lookup. One custodian, typically the server that created the file, is designated the *root* custodian or *home node*, and coordinates the file's replication and consistency management, as described below. When the root fails, another custodian is elected as root by a majority vote.

The focus of this paper is Swarm's consistency maintenance, so we briefly describe how files are replicated in Swarm, and refer the reader elsewhere [17] for further details on how Swarm creates cycle-free replica hierarchies in the face of node churn, optimizes the replica hiearchy based on observed network characteristics, and provides fault resilience.

Swarm caches files based on local access by clients. Figure 3 illustrates how the hierarchy is formed for file F2 as nodes successively cache it locally. Node distances in the figure are roughly indicative of their network distances (roundtrip times). When a Swarm server R wants to cache a file locally (to serve a local access), a Swarm server first uses the file's SWID to locate its custodians, e.g., via an external location service, like Pastry [14] or via a simple mechanism like hardwiring the IP address into the SWID, an approach we employ

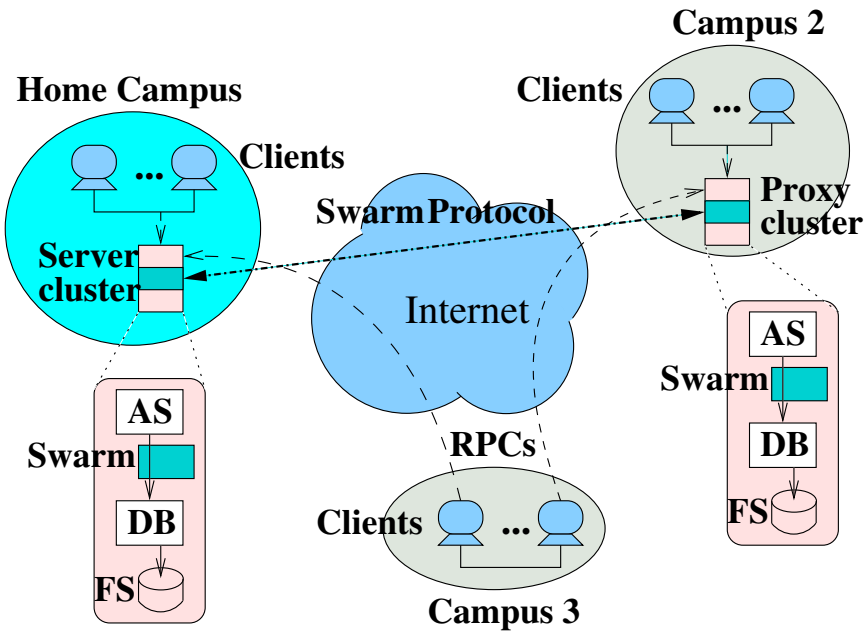

Figure 1: An enterprise application employing a Swarm-based proxy server. Clients in campus 2 access the local proxy server, while those in campus 3 invoke either server.

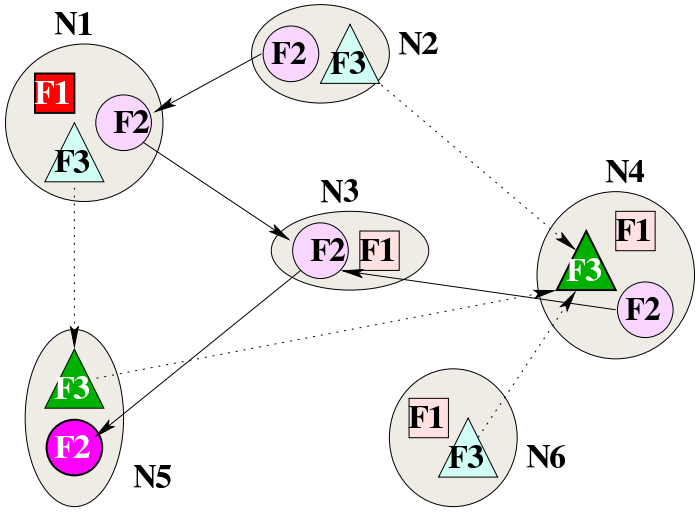

Figure 2: File replication in a Swarm network. Files F1..F3 are replicated at Swarm servers N1..N6. Permanent copies are shown in darker shade. F1 and F2 are homed at N1. F3 has two custodians: N4 and N5. Replica hierarchies are shown for F2 and F3 rooted at N5 and N4 respectively. Arrows indicate parent links.

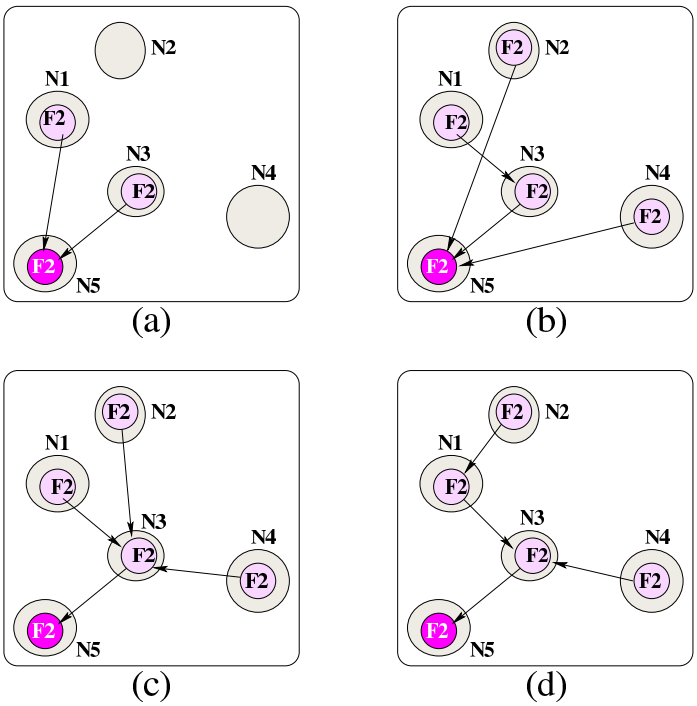

Figure 3: Replication of file F2. (a) N1 and N3 cache F2 from its home N5. (b) N2 and N4 cache it from N5; N1 reconnects to closer replica N3. (c) Both N2 and N4 reconnect to N3 as it is closer than N5. (d) Finally, N2 reconnects to N1 as it is closer than N3.

in our prototype. Swarm keeps track of these custodians, and then requests one of them (say P) to be its parent replica and provide a file copy, preferring those that can be reached via high quality (low-latency) links. Unless P is serving "too many" child replicas, it accepts R as its child, transfers the file contents, and initiates consistency maintenance (as explained in Section 5). P also sends the identities of its children to R, along with an indication if it has already reached its configured fanout limit. R augments its lookup cache with the supplied information. If P was overloaded, R remembers to avoid asking P for that file in the near future, otherwise R sets P as its parent replica. R repeats this parent election process until it has a valid file copy and a parent replica (the root custodian is its own parent). Even when a replica has a valid parent, it monitors its network quality to known replicas and reconnects to a closer replica, if found. This process forms a dynamic hierarchical replica network rooted at the root custodian like Blaze's file caching scheme [1], but avoids hops over slow network links when possible, like Pangaea [15]. When a replica loses contact with its parent, it reelects its parent in a similar manner to reconnect to the hierarchy. To guard against the root's temporary failure, the identities of the root's direct children are propagated to all replicas in the background, so reachable replicas can re-group into a tree until the root recovers.

5 Implementing Composable Consistency

To enforce composable consistency, each Swarm server has a consistency module (CC) that is invoked when clients open or close a file or when clients perform reads or updates within a session. Clients can indicate the desired consistency for file data accessed during a session. The CC module performs checks, interacting with peers when necessary, to ensure that the local file copy meets client requirements. Similarly, when a client issues updates, the CC module to take actions (i.e., propagating updates) to enforce consistency guarantees given by peer servers to their clients.

To succintly track concurrency control and replica synchronization guarantees (i.e., timeliness) given to client sessions, the CC module internally represents them by a construct called the *privilege vector (PV)*, described below. The CC module at a site can give a client access to a local replica without contacting its peers if the replica's PV indicates that the local replica's consistency is guaranteed to be at least as strong as the required by the client (e.g., if the PV indicates that the local copy is suitable for concurrent write access with 100ms staleness and the client can tolerate 200ms staleness). Peer CC modules exchange messages to learn of each other's PVs, obtain stronger PVs, ensure that their PVs are compatible, and push enough updates to preserve each other's PV properties. Although a Swarm server is responsible for detecting inaccessible peers and repairing replica

hierarchies, its CC module must continue to maintain consistency guarantees in spite of reorganization.

The rest of this section describes how the CC module tracks PVs and interacts with its peers to enforce CC options.

Privilege Vectors: The CC module internally represents concurrency control and replica synchronization (timeliness and strength) guarantees using a *privilege vector (PV)*. Associated with each replica of data is a *current privilege vector* (*currentPV*) that indicates the highest access mode and the tightest staleness and mod limit guarantees that can be given to local sessions without violating similar guarantees made at remote replicas.

A PV consists of four components that are independently enforced by different consistency mechanisms: an access mode, a hard time limit (HT), a hard mod limit (HM) and a soft time+mod limit (STM.[t,m]). By default, a file's root custodian starts with the highest PV ([WRLK, *, *, *] where * is a wildcard), whereas a new replica starts with the lowest PV ([RD, ∞, ∞, ∞]).

A replica obtains the PV required to grant local sessions by recursively issuing RPCs to its neighbors in the replica hierarchy. For each neighbor N, it remembers the relative PV granted to N (N.PVout) and obtained from N (N.PVin). The replica's currentPV is the lowest of the PVs it obtained from its neighbors. Since each replica only keeps track of the PVs of its neighbors, the PV state maintained at a replica is proportional to its fanout and not to the total number of replicas.

Core Consistency Protocol: The bulk of the CC module's functionality can be summarized in terms of two node-local operations:

- Pull(PV): Obtain the consistency guarantees represented by the PV from remote replicas.
- Push(neighbor N): Propagate outstanding updates to a neighbor replica N.

To implement these operations, Swarm's core consistency protocol employs two kinds of messages, namely, `get` and `put`, that are the inter-node counterparts of the above operations.

If a client initiates a session that requires stronger consistency guarantees than the local replica has (e.g., the client requests the file in WRLK mode but the local replica is read-only) or if the client requests a manual synch, the consistency module performs a *pull*

operation by issuing `get` messages to its neighbors, specifying the requested degree of consistency. The neighbors reply with `put` messages, which update the file's PV, and potentially its contents. For example, if the client requests the file in WRLK mode, the local consistency module waits until it is given exclusive write access to the file. To ensure fairness to client requests, concurrent pull operations for a particular file on a node are performed in FIFO order. A node issues `get` requests in parallel to its neighbors in the hierarchy, so pull latency grows logarithmically with the total number of replicas provided the hierarchy is kept dense by Swarm.

In contrast, a *push* operation is performed when there are updates available at a replica from either local clients or pushed from remote replicas that need to be passed along to other nodes in keeping with their consistency requirements. After the updates are applied locally, in keeping with the isolation options of local sessions, and are ready for propagation, in keeping with the originating session's visibility setting, the local consistency module sends `put` messages to its neighbors. The mod bounds and time bounds of remote replicas are maintained by propagating updates in accordance with their dependencies and ordering constraints. For example, if a node has guaranteed that it will help a neighboring node enforce a timeliness bound of 100ms, its consistency module may defer push'ing local updates to the neighbor until such time as deferring the push any longer will violate that guarantee, but no longer.

A Swarm server's CC protocol handler responds to a `get` message from a neighbor by first invoking a pull operation on the local CC module to obtain the requested PV locally. It then replies to the neighbor via a `put` message, transferring the requested privilege along with any pending updates, and updates the PV of its local replica to be compatible with the neighbor's new PV. By granting a PV, a replica promises to call back its neighbor before allowing accesses in its portion of the hierarchy that violate the granted PV's consistency guarantee. The recursive nature of the pull operation requires that the replica network be acyclic, since cycles in the network cause deadlocks. To avoid deadlock when parent and child replicas simultaneously issuing `get`s to each other, we allow a parent's pull request to bypass a child's `get` request to its parent.

When a `put` message arrives, the protocol handler blocks for ongoing updates to finish and local sessions with session-grained isolation to end. It then applies the incoming updates atomically, updates the file's PV, and makes a push request to the consistency module to trigger further update propagation (if necessary).

Leases: To reclaim PVs from unresponsive neighbors, a replica always grants non-trivial PVs higher than [wr, ∞, ∞, ∞] to its children as leases (time-limited privileges). The root node grants 60-second leases; other nodes grant slightly smaller leases than their parent lease to give them sufficient time to respond to lease revocation. A parent replica can

unilaterally revoke PVs from its children (and break its callback promise) after their finite lease expires, which lets it recover from a node holding a lease becoming unavailable. Child replicas that accept updates using a leased privilege must propagate them to their parent within the lease period, or risk update conflicts and inconsistencies. Leases are periodically refreshed via a simple mechanism whereby a node pings other nodes that have issued it a lease for any data (four times per lease period in our current implementation). Each successful ping response implicitly refreshes all leases issued by the pinged node that are held by the pinging node. If a parent is unresponsive, the node informs its own children that they cannot renew their lease.

When a child replica loses contact with its parent while holding a lease, it reconnects to the replica hierarchy and issues a special 'lease recovery' pull operation. Unlike a normal pull, lease recovery prompts an ancestor of the unresponsive old parent to immediately renew the lease, without waiting for the inaccessible node's lease to expire. This "quick reconnect" is legal because the recovering node has holds a valid lease on the data and thus has the "right" to have its lease recognized by its new parent. This mechanism enables replicas to maintain consistency guarantees in the face of node failures and a dynamically changing replica hierarchy.

Enforcing Replica Divergence Bounds: To enforce a hard time bound (HT), a replica R issues a pull to neighbors that are potential writers at most once every HT interval. A replica enforces a soft time bound (STM.t) by imposing a minimum push frequency (at least once per STM.t) on each of its neighbors. To enforce a modification bound of M unseen updates globally (HM and STM.m), a replica splits the bound into smaller bounds that are imposed on each of its neighbors that are potential writers. These neighbors may recursively split the bound to their children, which divides the responsibility of tracking updates across the replica hierarchy. A replica pushes updates whenever the number of queued updates reaches its local bound. If the mod bound is hard, the replica waits until the updates are applied and acknowledged by receivers before it allows subsequent updates.

Update Propagation: Swarm propagates updates (modified file blocks or operational updates) via `put` messages. Each Swarm server stores operational updates in FIFO order in a persistent *update log*. For the purpose of propagation, each update from a client session is tagged by its origin replica and a node-local version number to identify it globally. To ensure reliable update delivery, a replica keeps client updates in its log in the *tentative* state and propagates them via its parent until a custodian acknowledges the update, switches the update to the *saved* state, handles further propagation. When the origin replica sees that the update is now *saved*, it removes the update from its local log.

To identify what updates to propagate and ensure exactly-once delivery, Swarm maintains a *version vector (VV)* at each replica that indicates the latest update incorporated locally orig-

inating at every other replica. When two replicas synchronize, they exchange their VVs to identify missing updates. In general, the VV size is proportional to the total number of replicas, which could be very large (thousands), but since Swarm maintains replica trees and thus there is only one path between any two replicas in a stable tree topology, we can use compact neighbor-relative version numbers to weed out duplicate updates between already connected replicas. We exchange full version vectors only when a replica reconnects to a new parent, which occurs infrequently.

Ordering Concurrent Updates: When applying incoming remote updates, a replica checks if independent updates unknown to the sender were made elsewhere, indicating a potential conflict. Conflicts are possible when clients employ concurrent mode write (WR) sessions. For 'serially' ordered updates, Swarm forwards them to the root custodian to be applied sequentially. For 'unordered' updates, Swarm applies updates in their arrival order, which might vary from one replica to another. For 'totally' ordered updates, Swarm relies on a conflict resolution routine to impose a common global order at all replicas. This routine must apply the same update ordering criterion at all replicas to ensure a convergent final outcome. Swarm provides default resolution routines that reorder updates based on their origination timestamp (*timestamp order*) or by their arrival order at the root custodian (*centralized commit order*. The former approach requires Swarm servers to loosely synchronize their clocks via protocols such as NTP. The latter approach is similar to Bayou [3] and uses version vectors.

To enforce semantic ordering constraints (e.g., atomic or causal), Swarm tags each update with the ordering constraint of the issuing session. It uses the local update log to determine in what order it must propagate or apply updates to satisfy these ordering guarantees. Atomically grouped updates are always propagated and applied together.

6 Evaluation

In Section 6.1, we show how Swarm's CC implementation of close-to-open semantics exploits 'nearby replicas' (similar to Pangaea [15]) to outperform Coda's client-server implementation in a sequential file sharing (roaming) scenario. In Section 6.2 we demonstrate how the performance and scalability of a replicated BerkeleyDB database varies under five consistency mechanisms that range from 'strong' (appropriate for a conventional database) to 'time-based' (appropriate for typical directory services). Swarm offloaded the hard problems of replication and consistency management from both applications, thereby simplifying their implementation while providing them with a rich set of consistency choices.

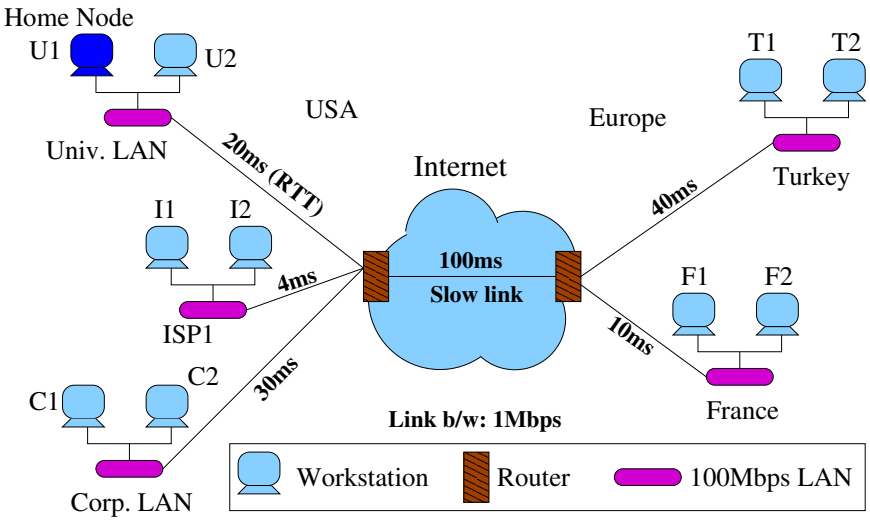

Figure 4: Network topology for Swarmfs roaming experiment.

For all experiments, we used the Emulab Network Testbed [4] to emulate WAN topologies among clustered PCs. The PCs have 850Mhz Pentium-III CPUs with 512MB of RAM and run Redhat Linux 7.2. Swarm servers run on each PC as user-level processes and store files in a single directory in the local file system, using the SWID as the filename. The Swarm replica fanout was configured to a low value of 4 to induce a multi-level hierarchy.

6.1 Swarmfs: A Flexible Distributed File System

We evaluate a Swarm-based file system (Swarmfs) on a synthetic roaming benchmark across an emulated non-uniform WAN network shown in Figure 4. In this benchmark, we model collaborators at a series of locations accessing shared files, one location at a time. This type of 'sequential file sharing' is representative of mobile file access or workflow applications where collaborators take turns updating shared documents or files. We compare Swarmfs to Coda employing both weak (coda-w) and strong connectivity modes (coda-s). Like Pangaea, and unlike Coda, Swarm provides low-latency by treating replicas as peers for consistency enforcement. Unlike Pangaea, Swarmfs guarantees close-to-open consistency. To support close collaboration or near-simultaneous file access, users require close-to-open consistency semantics even under weak connectivity.

The topology modeled consists of five widely distributed campuses, each with two machines on a 100Mbps LAN. The node U1 (marked 'home node') initially stores the Tcl-8.4

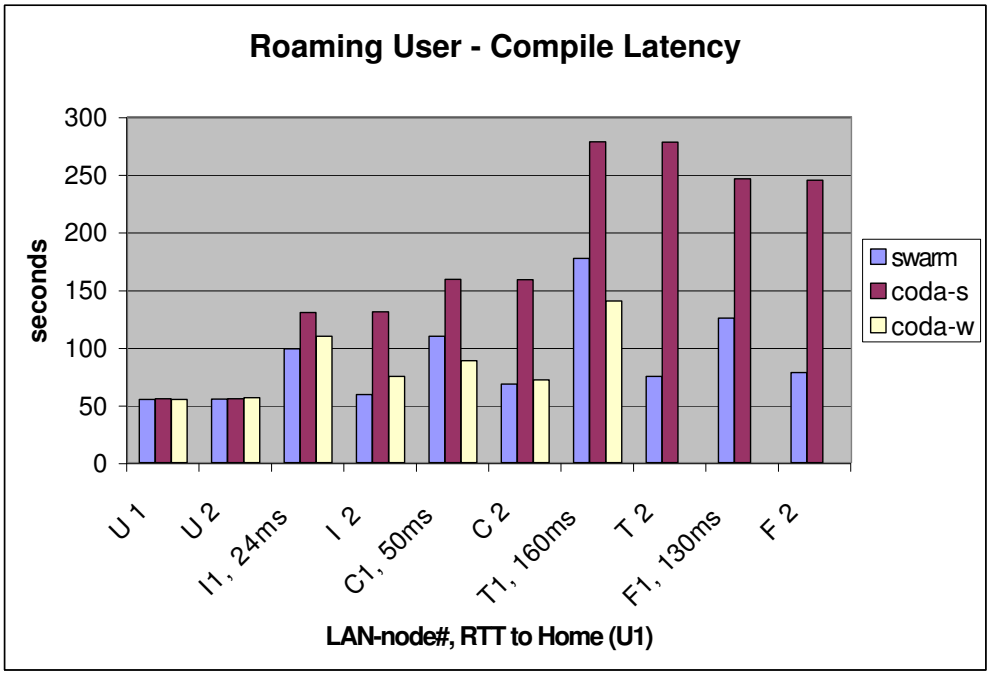

Figure 5: Roaming File Access: Swarmfs pulls source files from nearby replicas. Strong-mode Coda correctly compiles all files, but exhibits poor performance. Weak-mode Coda performs well, but generates incorrect results on the three nodes farthest from the server (U1).

source files, 163 files with a total size of 6.7MB. We run Swarmfs servers and Coda clients on all nodes. The Coda server runs on U1.

In our synthetic benchmark, clients at various nodes sequentially access files. each client modifies one source file, compiles the Tcl-8.4 source tree, and then deletes all object files. These operations represent isolated updates, intensive file-based computation, and creating and deleting a large number of temporary files. Clients on each node in each campus (in the order University (U) $\rightarrow$ ISP1 (I) $\rightarrow$ Corporate (C) $\rightarrow$ Turkey (T) $\rightarrow$ France (F)), perform the edit-compile-cleanup operation one after another.

Figure 5 shows the compilation times on each node. Since Swarm creates efficient replica

hierarchies and acquires files from nearby replicas (e.g., from another node on the same LAN, or, in the case of France, from Turkey), it outperforms Coda's client-server implementation, which always pulls files from server U1.

In weak connectivity mode (coda-w), Coda guarantees only eventual consistency, which causes incorrect behavior starting at node T2. In this case, coda-w's trickle reintegration causes the client on T1 to eagerly push huge object files to U1, thereby clogging its network link and delaying notification of subsequent file deletions. By the time T2 sees T1's file deletion operations, it has already started its compile and used stale object files. Coda's strong connectivity mode provides close-to-open semantics but incurs double the latency of Swarmfs because of write-throughs to the server (U1). Both these problems occur because Coda servers never pull updates from clients, but rather clients must push updates to the server. Swarm avoids this by employing the same pull mechanism uniformly at all replicas.

6.2 SwarmDB: Replicated BerkeleyDB

Popular databases (e.g., mySQL, Oracle, and BerkeleyDB) predominantly employ master-slave replication (if any) across the wide area due to its simplicity; read-only replicas are deployed near clients to scale query performance, but updates are applied at a central master site to ensure serializability. For applications that can handle concurrent updates (e.g., many directory services), master-slave replication is overly restrictive and does not scale or exploit regional locality. By using Swarm to implement database replication, we can choose on a per-client basis how much consistency is required. High throughput can be achieved when the consistency requirements are weaker, as in a directory service [10], while the same code base can be used to provide a strongly consistent database when required.

We augmented the BerkeleyDB embedded database library [16] with replication support as a wrapper library called *SwarmDB*. A typical SwarmDB-based application is shown in Figure 1. SwarmDB stores a BerkeleyDB database in its native format in a Swarm file and intercepts BerkeleyDB update operations to invoke them as operational updates on the local Swarm server via a SwarmDB plugin. SwarmDB-based application instances access the database via Swarm servers spread across a WAN that cache the database locally.

We measure SwarmDB's read and write throughput under a full-speed update-intensive workload when employing the five distinct flavors of consistency semantics shown in Table 2. We compare the performance of the five flavors of SwarmDB against BerkeleyDB's client-server (RPC) implementation. The consistency flavors (listed from strongest to weakest) are: (1) *locking writes and optimistic reads*, where writes are serialized via locks before being propagated, (2) *master-slave writes and optimistic reads*, where all writes

Consistency Semantics	CC options
locking writes	WRLK
master-slave writes	WR, serial
close-to-open rd, wr	RD/WR, time=0, hard
time-bounded rd, wr	time=10, hard
optimistic/eventual rd, wr	RD/WR, time=0, soft

Table 2: Consistency flavors employed for Replicated BerkeleyDB and the CC options to achieve them. The unspecified options are set to [RD/WR, time=0, mod=∞, soft, no semantic deps, total order, pessimistic, session visibility & isolation].

are serially ordered at the root of the replica hierarchy before propagation, (3) *close-to-open consistency*, (4) *time-bounded staleness*, where data is synched before access if more than a threshold time has passed since the last synch, and (5) *optimistic writes and reads*, where writes are performed locally before being propagated to other replicas. Our synthetic benchmark creates and populates a BerkeleyDB database with 1000 key-value pairs inside a Swarm file. The database size does not affect performance, except during startup,since we employ operational updates, where Swarm replicates the entire database file as a single consistency unit and propagates operations instead of changes.

We run the benchmark on 2 to 48 nodes. Nodes are each connected by a 1Mbps, 10-msec delay WAN link to a backbone router, which implies a 40ms roundtrip between any two nodes. Each server executes 10,000 random operations at full-speed, i.e., no think time. The operation mix consists of 5% adds, 5% deletes, 20% updates, 30% lookups, and 40% cursor-based scans. Reads (lookups and cursor-based scans) are performed directly on the database file, while writes (adds, deletes, and updates) are sent to the local Swarm server by the SwarmDB library. Each operation opens a Swarm session on the database file in the appropriate mode, performs the operation, and closes the session.

Figures 6 and 7 show the average throughput observed per replica for reads and writes. In addition to the SwarmDB results, we present baseline performance when directly operating on a database file stored in the local file system (*local*), when invoking RPCs to a colocated berkeleyDB server (*rpclocal*), and when accessing a local Swarm-based database file with no sharing (*Swarm local*). Swarm local represents the best throughput possible using SwarmDB on top of our Swarm prototype. The high cost of IPC between the client and server account for the performance degradation of *rpclocal* and *Swarm local* compared to *local*.

Figure 6 shows that read throughput scales well when we request soft (push-based) or time-based consistency guarantees, but not when we request hard (firm pull-based) guarantees, as expected. Due to the update-intensive nature of the workload, there is almost

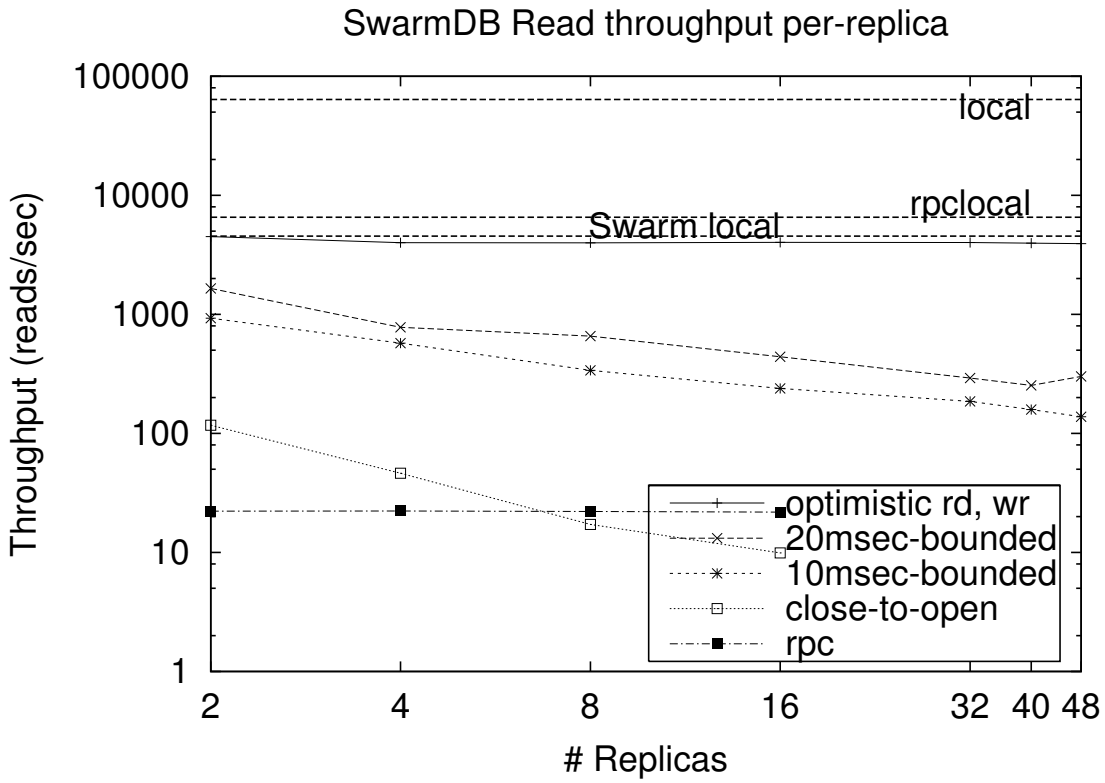

Figure 6: SwarmDB Per-replica Read Throughput

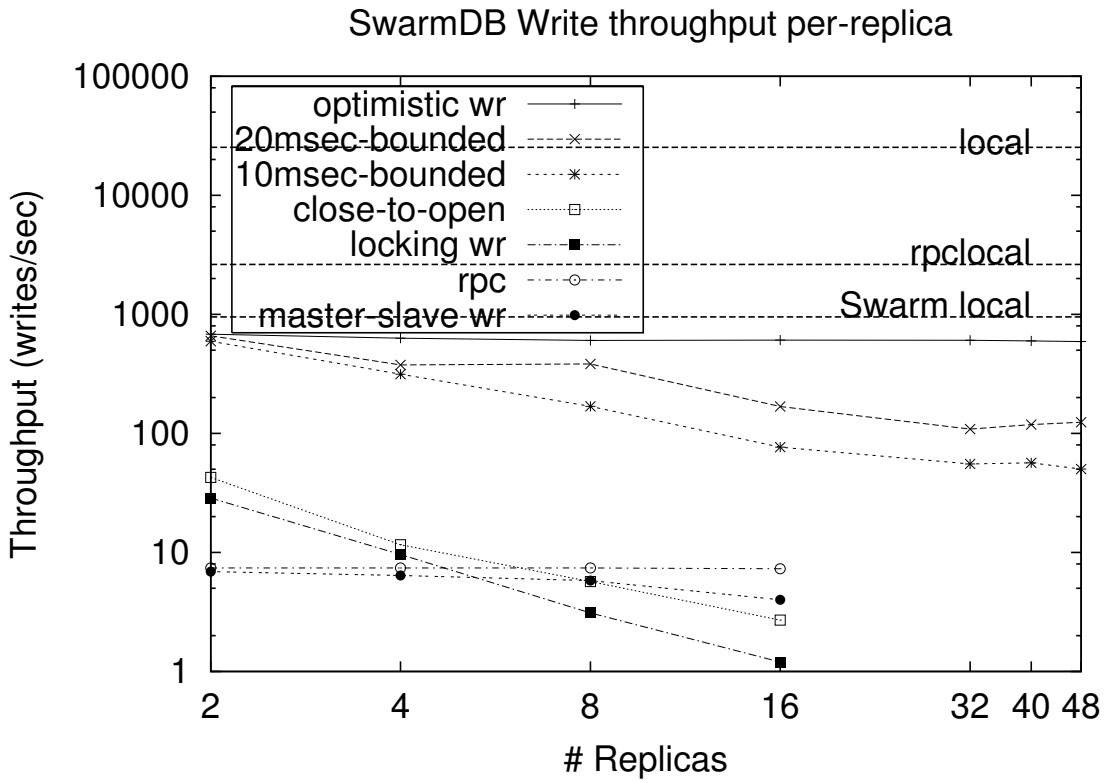

Figure 7: SwarmDB Per-replica Write Throughput

always a write in progress somewhere in the system. Thus the strict pull-based schemes are constantly pulling updates across WAN links, suffering the same high latency that the RPC-based solution incurs. As a result, close-to-open and strong consistency do not scale beyond 16 replicas, given the high degree of write-sharing. Eventual consistency scales well to large replica sets even when pushing updates eagerly, because Swarm enables the DB plugin to remove self-canceling updates. Tolerating even a small amount of staleness (10ms) significantly improves read and write performance over close-to-open consistency, because the cost of synchronization over the wide area is very high, and amortizing it over multiple operations has substantial latency benefit.

In summary, different consistency options provide vastly different semantics and performance characteristics for the same workload. Composable consistency enables an application to choose the right semantics based on its specific need at hand.

7 Conclusions

In this paper we proposed a new way to structure consistency management for P2P sharing of mutable data, called composable consistency. It splits consistency management into design choices along several orthogonal dimensions and lets applications express their consistency requirements as a vector of these choices on a per-access basis. The design choices are orthogonal and can be combined in various ways to yield a rich collection of semantics, while enabling an efficient implementation. We outlined an implementation of the model in a pervasive peer replication environment spanning non-uniform networks. Our evaluation showed how composable consistency is both expressive and practical.

References

[1] M. Blaze. *Caching in Large Scale Distributed File Systems*. PhD thesis, Princeton University, 1993.

[2] L. Cox and B. Noble. Fast reconciliations in Fluid Replication. In *Proc. 21st Intl. Conference on Distributed Conputing Systems*, Apr. 2001.

[3] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The Bayou architecture: Support for data sharing among mobile users. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, Dec. 1994.

[4] Emulab. `http://www.emulab.net/`, 2001.

[5] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–82, Feb. 1988.

[6] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proc. 13th Symposium on Operating Systems Principles*, pages 213–225, Oct. 1991.

[7] N. Krishnakumar and A. Bernstein. Bounded ignorance: A technique for increasing concurrency in a replicated system. *ACM Transactions on Data Base Systems*, 19(4), Dec. 1994.

[8] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4), 1992.

[9] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The Objectstore database system. *Communications of the ACM*, Oct. 1991.

[10] Microsoft Corp. Active directory (in windows 2000 server resource kit). Microsoft Press, 2000.

[11] A. Muthitacharoen, B. Chen, and D. Mazieres. Ivy: A read/write peer-to-peer file system. In *Proc. 5th Symposium on Operating System Design and Implementation*, Dec. 2002.

[12] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, 1988.

[13] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G. Popek. Resolving file conflicts in the Ficus file system. In *Proceedings of the Summer Usenix Conference*, 1994.

[14] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. 18th Symposium on Operating Systems Principles*, 2001.

[15] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In *Proc. 5th Symposium on Operating System Design and Implementation*, pages 15–30, 2002.

[16] Sleepycat Software. The BerkeleyDB database. `http://sleepycat.com/`, 2000.

[17] S. Susarla and J. Carter. Flexible consistency for wide area peer replication. Technical Report UUCS-04-016, University of Utah School of Computer Science, Oct. 2004.

[18] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proc. 4th Symposium on Operating System Design and Implementation*, Oct. 2000.