Middleware Support for Locality-aware Wide area Replication

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

UUCS-04-017

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

November 18, 2004

Abstract

Coherent wide-area data caching can improve the scalability and responsiveness of distributed services such as wide-area file access, database and directory services, and content distribution. However, distributed services differ widely in the frequency of read/write sharing, the amount of contention between clients for the same data, and their ability to make tradeoffs between consistency and availability. Aggressive replication enhances the scalability and availability of services with read-mostly data or data that need not be kept strongly consistent. However, for applications that require strong consistency of write-shared data, you must throttle replication to achieve reasonable performance.

We have developed a middleware data store called *Swarm* designed to support the wide-area data sharing needs of distributed services. To support the needs of diverse distributed services, Swarm provides: (i) a failure-resilient proximity-aware data replication mechanism that adjusts the replication hierarchy based on observed network characteristics and node availability, (ii) a customizable consistency mechanism that allows applications to specify allowable consistency-availability tradeoffs, and (iii) a contention-aware caching mechanism that monitors contention between replicas and adjusts its replication policies accordingly. On a 240-node P2P file sharing system, Swarm's proximity-aware caching and replica hierarchy maintenance mechanisms improve latency by 80%, reduce WAN bandwidth consumed by 80%, and limit the impact of high node churn (5 node deaths/sec) to roughly one-fifth that of random replication. In addition, Swarm's contention-aware caching mechanism outperforms RPCs and static caching mechanisms at all levels of contention on an enterprise service workload.

1 Introduction

Given the pervasive use of the Internet to access remote services, centralized servers cannot always deliver high performance and high availability to end users. Clients frequently need to access services over wide area networks that are characterized by long delays that vary due to congestion and cross-traffic and that are prone to occasional failures. Replicating[1] service data close to end-users is a common technique to hide this network variability. Although caching can improve service quality, its effectiveness depends on application characteristics such as the application's consistency requirements, the frequency of read/write data sharing, the amount of contention between clients for the same data, and the application's ability to make tradeoffs between consistency and availability.

Many replication techniques have been proposed to improve wide area service availability and/or performance for specific applications and environments [9, 4, 14, 17]. For example, aggressively replicating data wherever data is accessed has been shown to improve the availability and scalability of wide-area file access [17]. Unfortunately, not all applications benefit from aggressive replication. For example, while database designers often replicate data for scalability within clusters [13], they often restrict remote users to using RPCs to avoid the high cost of replica synchronization. In general, when clients contend for strongly consistent access to write-shared data, aggressive caching performs poorly.

Several consistency models (e.g., TACT [21]) improve service quality by selectively weakening the consistency of data delivered to users. For example, applications like directory services can employ weak consistency models to avoid frequent data synchronization. However, for applications like online auctions, relaxed data consistency can lead to incorrect behaviors, e.g., selling the same item to multiple customers. It is easy to enforce strong consistency by requiring all accesses to be performed via RPCs to a central database, but this design can lead to poor performance and is unnecessary if there is little contention for data.

In this paper, we present a middleware data store called *Swarm* that supports the wide-area data caching needs of a variety of wide-area distributed services. It relieves application writers of the burden of implementing their own replication and consistency mechanisms. Swarm builds failure-resilient dynamic replica hierarchies to manage large numbers (thousands) of replicas across wide-area networks. By default, Swarm aggressively replicates shared data based on observed access patterns, but it dynamically restricts replication when it observes high contention. Swarms's replication mechanism dynamically monitors the

[1] In this paper, we use the term *replication* to mean caching for availability, not *first-class* replication [8] for data reliability.

network quality between clients and replicas and reorganizes the replica hierarchy to minimize the use of slow links, thereby reducing latency and saving WAN bandwidth [17]. We refer to this feature of Swarm as *proximity-aware caching*. Like TACT[21], Swarm allows applications to tune consistency to their availability needs. Swarms supports strong consistency, close-to-open consistency, bounded inconsistency (based on staleness and unseen writes), and best-effort eventual consistency. Finally, Swarm's replica hierarchy maintenance mechanism is resilient to network and node failures, and handles node churn [10] gracefully.

Swarm makes three specific contributions to the field of wide area replication. First, it provides a novel replication mechanism that supports a broader variety of consistency semantics than existing large-scale systems. Second, we demonstrate how tracking replica consistency invariants via concise *privilege vectors* and employing *proximity-aware caching* enables scaling. Finally, Swarm dynamically tracks read-write data contention and helps applications switch between aggressive caching (data-shipping) and RPCs (function-shipping), a feature we call *contention-aware caching*. During periods of high contention, Swarm elects one replica, typically the one that gets the most accesses, as *master* and forwards all client requests to that replica to avoid thrashing. When contention subsides, it resumes aggressive peer caching to reduce latency.

This paper focuses on Swarm's replica hierarchy maintainence, proximity-aware caching, and contention-aware caching mechanisms. We do not discuss in detail Swarm's flexible consistency model, which is discussed elsewhere [19], and the current Swarm prototype does not consider security or authentication, nor does it implement first-class replication. Proximity-aware caching (i.e., the ability to cache files from nearby replicas) improves P2P file access latency on a 240-node system by 80% and reduces the WAN bandwidth consumed by 75% compared to random replication. Even with a high node churn of 5 nodes/sec, performance degrades by at most 100% compared to a 500% slowdown using a non-adaptive mechanism. When we evaluated Swarm's contention-aware caching mechanism using a synthetic enterprise object-caching application running a benchmark similar to TPC-A, we found that it is able to support strong consistency and outperform both aggressive caching and function-shipping at all levels of contention.

We present an overview of Swarm in Section 2. We describe Swarm's replication, consistency management, and contention-aware caching mechanisms in Sections 3, 4, and 5, respectively. We present our evaluation of Swarm's replication mechanisms in Section 6. Finally, we discuss prior work and draw conclusions in Sections 7 and 8.

2 Swarm Overview

Swarm is a distributed file store organized as a collection of peer 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. A special metadata block similar to a Unix inode holds the file's attributes, e.g., its size and consistency options. Swarm servers locate files by their SWIDs, cache them as a side-effect of local access, and maintain consistency according to consistency options settable on a per-file-access basis. 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. This enables users in a LAN to synchronize with each other directly without going through a far away replica.

Interface: Swarm exports a 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 [9]. 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 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 an entire file as a single consistency unit.

Consistency Options: Swarm clients can indicate their consistency needs for a file when opening a session. Swarm supports four flavors of consistency that cover a variety of application needs: (1) *strong consistency*, which provides traditional concurrent-read-exclusive-write semantics, (2) *close-to-open*, which allows concurrent writes but ensures that clients see the latest data at every `open()`, (3) *bounded inconsistency*, which limits how stale data can be in terms of either time or the number of unseen updates [21], or (4) *eventual consistency*, which provides optimistic replication with soft time and mod bounds. Swarm can enforce distinct semantics for different clients on the same data. The first three flavors provide hard consistency guarantees that are enforced by synchronously pulling data when necessary.

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). As described in Section 3, the local Swarm server interacts with other Swarm servers to acquire and maintain a locally cached copy of the file with the specified consistency. Figure 1 illustrates how a database-backed auction service could be organized to employ Swarm-based wide area caching proxies. Clients can access any of the available server clusters, which are functionally identical. The 'DB' represents an unmodified replication-oblivious database backend (such as BerkeleyDB [18] or MySQL), the 'AS' represents auction-specific logic, and the 'FS' provides local storage. To use Swarm for replication, the auction service stores its database in Swarm file(s). The AS opens a Swarm session on a local database copy and performs queries and updates. The AS updates the database either by accessing its file blocks via Swarm (absolute updates) or by issuing operational updates to the local Swarm server. In particular, neither the AS nor the DB need to implement wide area replication and consistency management logic, simplifying their implementation. A distributed file system could be similarly organized, but without the DB component.

Contention-Aware Caching: An application can use Swarm's contention-aware caching mechanism to choose between data-shipping and function-shipping for each session as follows. As part of its per-file consistency attributes, each file has a *soft* and a *hard caching threshold* that indicate the minimum number of local accesses a replica must see before it can switch to aggressive caching. The soft and hard limits provide hysteresis to prevent oscillations. A zero threshold forces aggressive peer caching, while a high value forces centralization. Ideally, the middleware should determine these limits automatically based on the ratio of the cost of caching data to that of performing RPCs remote servers, but our prototype requires that the thresholds be specified by the application.

When opening a Swarm session on a replica, the AS can ask Swarm to select a remote proxy server *master* site and redirect local application-level operations to this master to avoid contention. If Swarm suggests a remote site, the AS issues an application-level RPC to the AS near that site via external means. Swarm tracks the number of such sessions redi-

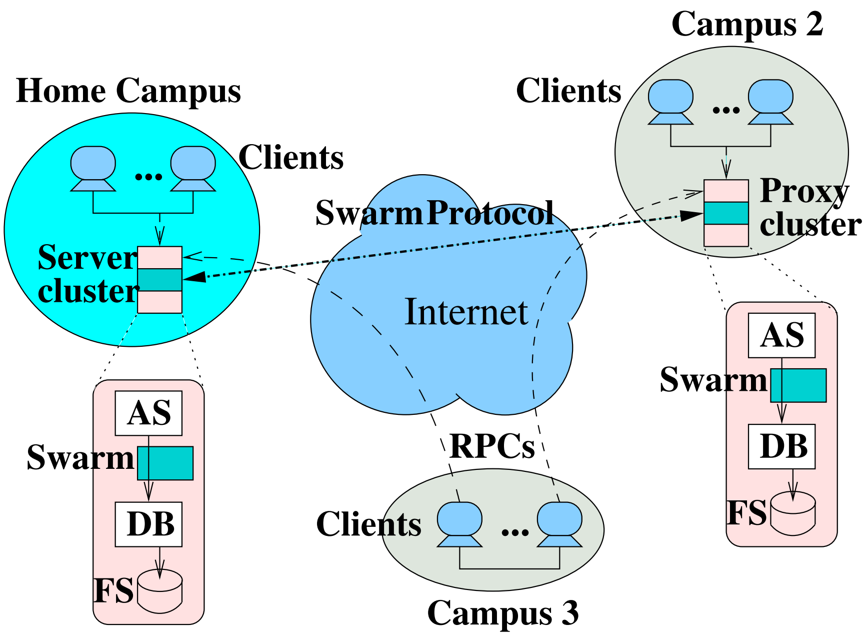


Figure 1: An enterprise application employing a Swarm-based proxy server. Clients in campus 2 access the local proxy server, while those in campus 3 invoke either server.

rected internally to guide its future master elections. However, the AS need not query for a master, in which case Swarm performs aggressive caching. Thus, each AS chooses whether to employ contention-aware or aggressive caching, but Swarm determines whether to use data-shipping or function-shipping for individual operations based on its global knowledge of contention.

Overview of Swarm Operation: Figure 2 shows the internal structure of a Swarm server with its major modules and their flow of control. A Swarm file access request (e.g., open) is handled by the Swarm server's session management module, which invokes the replication module to fetch a consistent local copy in the desired access mode. The replication module creates a local replica after locating and connecting to another available replica nearby. It then invokes the consistency module to bring the local copy to the desired consistency by interacting with other replicas via pull and push operations.

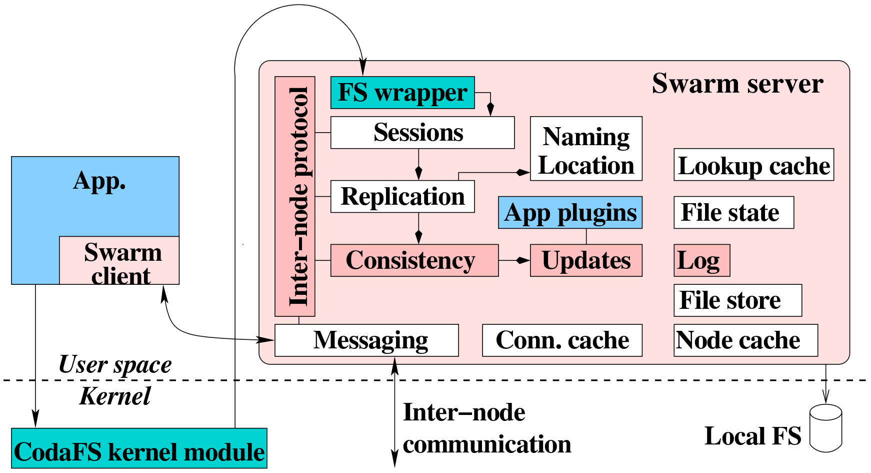


Figure 2: Structure of a Swarm server and client process.

3 Replication

Having described Swarm's architecture and usage, we now turn to describing Swarm's replication mechanisms which are the focus of this paper. Swarm's replication design must meet three goals: it must enable the enforcement of the diverse consistency guarantees listed earlier, provide low latency access to data despite large-scale replication, and limit the load imposed on each replica to a manageable level.

Replica Hierarchy: The Swarm servers caching a particular file dynamically organize themselves into an overlay *replica hierarchy* for network-efficient communication and scaling. They impose a user-configurable *replica fanout* (#children, typically 4 to 8) to limit the amount of replica maintenance traffic handled by a replica. To support strong flavors of consistency across non-uniform links, Swarm employs a consistency protocol based on recursive messaging along the links of the replica hierarchy. Hence it requires that the replica network be acyclic. Figure 3 shows a Swarm network of six servers with hierarchies for two files. One or more servers maintain permanent copies of a file, and are called its *custodians*. 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, other custodians elect one among themselves as root by a majority vote. However, our prototype implementation does not support multiple custodians since there are well-known fault-tolerant replication solutions [8]. Our research focus is rather on scalable caching and consistency mechanisms.

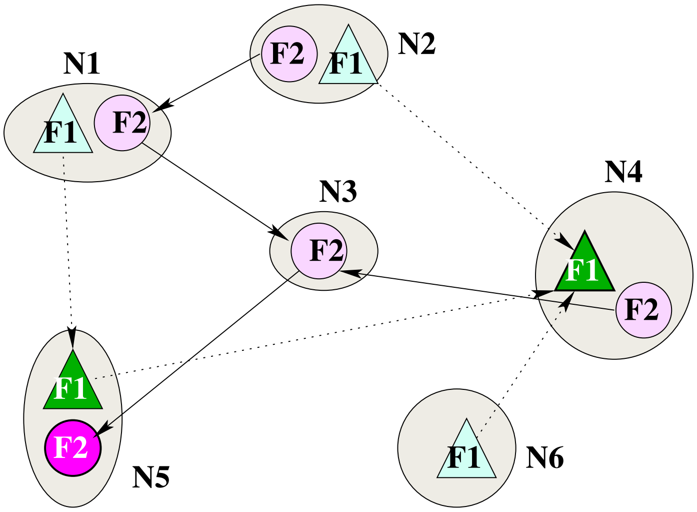


Figure 3: File replication in a Swarm network. Files F1 and F2 are replicated at Swarm servers N1..N6. Permanent copies are shown in darker shade. F1 has two custodians: N4 and N5. F2 is homed at N1. Arrows indicate parent links of the file replica hierarchies.

In the rest of this section, we describe how Swarm creates and destroys replicas and maintains dynamic cycle-free hierarchies and how a large number of Swarm nodes discover and track each other's accessibility and connection quality.

Creating Replicas: A Swarm server caches a file locally when clients request access to it. To do so, it first uses the file's SWID to locate its custodians, e.g., via an external location service, like Pastry [15] or via a simple mechanism like hardwiring the home IP address into the SWID, an approach we employ in our prototype. Swarm keeps track of these custodians in a local *lookup cache*, and then requests one of them (say P) to be R's 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, P accepts R as its child, transfers the file contents, and initiates consistency maintenance as explained below. P also sends the identities of its children to R, along with an indication if it has already reached its configured fanout (#children) 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 in the background, if found. R detaches from current parent only after attaching successfully to a new parent.

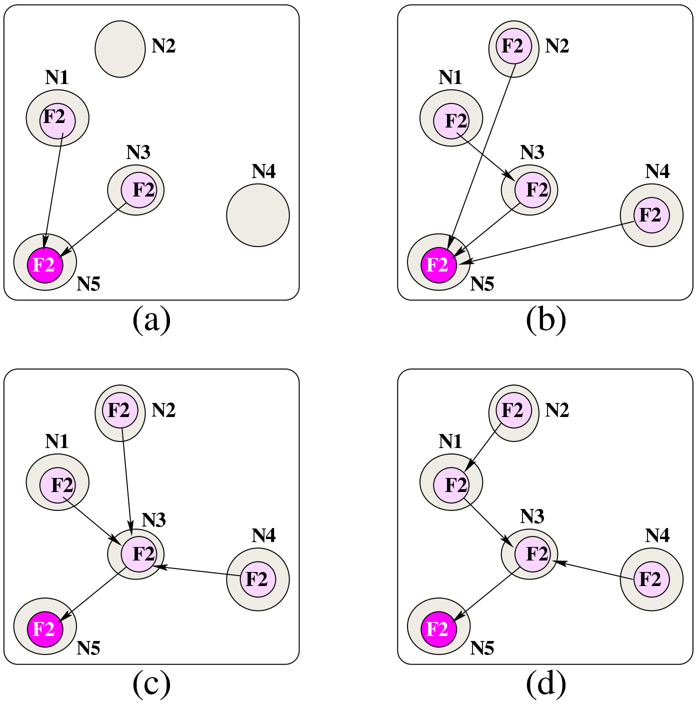


Figure 4: Replication of file F2. (a) N1 and N3 cache F2 from its home N5. (b) N2 and N4 cache it from N5; N1 reconnects to closer replica N3. (c) Both N2 and N4 reconnect to N3 as it is closer than N5. (d) Finally, N2 reconnects to N1 as it is closer than N3.

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 [17]. To keep the hierarchy dense, a replica with fewer children than its fanout limit periodically advertises itself to its subtree as a potential parent. Thus, its descendants can connect to it and move up the hierarchy if they find it to be nearer. We believe that a fanout of 4 to 8 provides a reasonable balance between load due to many children and extra hops needed to traverse a deep hierarchy. We evaluated Swarm with hierarchies up to 4 or 5 levels deep and a fanout of 4. Since replicated performance is primarily sensitive to number of hops and hence depth, we expect Swarm to gracefully handle a fanout of 8, enabling it to scale to thousands of replicas with hierarchies of similar depth.

Avoiding Cycles. The above parent election scheme could form cycles. Hence before accepting a new parent replica, an interior replica (i.e., one that has children) actively detects cycles by propagating a root probe request along parent links. If a probe gets forwarded to its originator, it means that accepting the new parent will form a cycle. If the probe reaches a custodian's children, they reply directly to the initiator to reduce load on custodians including the root. Leaf replicas do not accept new children while they themselves are electing their parent. Hence they need not probe for cycles, nor do they form primitive cycles. Our evaluation in Section 6.1 reveals that the overhead of root probes is low even under heavy node churn.

Figure 4 illustrates how file F2 ends up with the replica hierarchy shown in Figure 3, assuming node distances in the figure roughly indicate their network roundtrip times. Initially, the file is homed at node N5, its only custodian. Nodes N1 and N3 cache the file, making N5 their parent (Figure 4a). Subsequently, nodes N2 and N4 also cache it directly from N5, while N1 reconnects to its nearby replica N3 as its parent (Figure 4b)[2]. Since N5 informs its new children of their sibling N3, they connect to N3, which is nearer to them than N5 (Figure 4c). At that time, N2 gets informed of the nearer replica at N1. Thus, N2 makes N1 its parent replica for file F2 (Figure 4d).

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. When a replica gets partitioned, it continues to provide the desired consistency to local clients and child replicas. We describe how Swarm handles node and network failures below.

Retiring Replicas: Each Swarm server (other than a custodian) tracks local usage of

[2] Note that N3 could have reconnected to N1 instead. If both try connecting to each other, only one of them succeeds, as one has to initiate a root probe which detects the cycle.

cached file copies and evicts least recently used ones after propagating their updates to neighboring replicas and informing them of the replica's departure. The orphaned child replicas elect a new parent as described above. However, before a custodian (including root) can retire, it must inform the root node. as custodians must participate in majority voting for root election. When the root node itself wants to retire, it first transfers the root status to another custodian (currently, one with lowest IP address) via a two-phase protocol and becomes its non-custodian child. Thus, a file's custody can migrate among nodes.

Node Membership: Swarm servers learn about each other's existence as a side-effect of looking up SWIDs. Each Swarm server communicates with another via a TCP connection to ensure reliable, ordered delivery of messages. Since Swarm servers form replica hierarchies independently for each file, a Swarm server that caches a lot of files might need to communicate with a large number of other Swarm servers, potentially causing scaling problems. To facilitate scalable communication, each Swarm server keeps a fixed-size LRU cache of live connections to remote nodes (called a *connection cache*), and a larger persistent *node cache* to remember the status of remote nodes contacted recently. A replica need not maintain an active connection with its neighboring replicas all the time, enabling a Swarm server's scalable handling of a large number of files and other Swarm servers.

A Swarm server assumes that another node is reachable as long it can establish a TCP connection to it. If TCP shuts down the connection due to a communication error (as opposed to a graceful close by peer), the Swarm server infers that the other node is unreachable and avoids it in subsequent lookups and replica reconnection attempts for a timeout period. A Swarm node treats a busy neighbor differently from unreachable neighbor by employing longer timeouts for replies from a connected node to avoid prematurely reacting to transient overload.

Network Economy: To aid in forming replica hierarchies that utilize network efficiently, Swarm servers continually monitor the quality of network links to each other and persistently cache this information. We currently employ roundtrip time (RTT) as our quality metric for links, but can also encode link bandwidth and lossiness. In our current design, each replica decides for itself which other replicas are close to it by pinging them individually (piggybacked on other Swarm messages if possible). The ping traffic is initially high when new nodes join the Swarm network, but quickly subsides and is unaffected by nodes going up and down. However, Swarm can easily adopt better RTT estimation techniques [6] as they become available.

Failure Resilience: Swarm treats custodians (including root) as first-class replicas for reliability against permanent loss, and employs a majority voting algorithm [8] to provide Byzantine fault-tolerance. Hence for high availability under strong consistency, custodian copies must be small in number (typically 4), and hosted on well-connected machines.

However, Swarm can handle a large number (thousands) of secondary replicas and their failures due to its dynamic hierarchy maintenance. Swarm employs leases to recover access privileges from unreachable non-custodian replicas as explained in Section 4. Swarm handles transient root node failure as explained earlier in this section. When a replica permanently fails, only those updates originating in it that have not reached another replica are lost. To protect against such loss, a replica periodically pushes updates to custodians regardless of consistency guarantees.

4 Consistency Management

In this section, we outline how Swarm uses its hierarchical replication to enforce diverse consistency semantics. Each Swarm server has a consistency module (CM) that is invoked when clients open or close a file or when clients perform reads or updates within a session. The CM performs checks, interacting with replica neighbors (parent and children) via *pull* operations as described below, to ensure that the local file copy meets client requirements. Similarly, when a client issues updates, the CM propagates them via *push* operations to enforce consistency guarantees given by peer servers to their clients.

To succintly track concurrency control and replica divergence guarantees (i.e., time and mod bounds) given to client sessions, the CM internally represents them by a construct called the *privilege vector (PV)*. The CM can allow a client to access local replica without contacting peers if the replica's consistency indicated by its PV is at least as strong as required by the client. For example, if a client requires 200ms staleness and the PV guarantees a max staleness of 100ms, no pull is required. CMs at neighboring replicas in the hierarchy exchange PVs based on client consistency demands, ensure that PVs do not violate guarantees of remote PVs, and push enough updates to preserve each other's PV guarantees. Although a Swarm server is responsible for detecting inaccessible peers and repairing replica hierarchies, its CM must continue to maintain consistency guarantees in spite of reorganization. To recover from unresponsive peers, the CMs at parent replicas grant PVs to children as leases, as explained below.

Privilege Vectors: A PV consists of four components that are independently enforced by different consistency mechanisms: an access mode (mode), a hard time/staleness limit (HT), a hard mod limit (HM) and a soft time+mod limit (STM.[t,m]). There are four access modes: locking read (RDLK) and write (WRLK) modes, and Unix-style file read (RD) and write (WR) modes that are non-blocking. PVs are partially ordered by a *PVincludes* relation defined in Figure 5 that indicates how to compare the strength of guarantee provided by each of its components. By default, a file's root custodian starts with the highest PV

```
PVincludes(PV1, PV2) /* $\supseteq$ relationship */
{
 /* checks if PV1 gives PV2's consistency guarantees */
 /* WRLK > WR > RD; WRLK > RDLK > RD */
 if (PV1.mode == WRLK) or
    ((PV1.mode == RDLK) and (PV1.mode >= PV2.mode))
   return TRUE;
 else if (PV1.mode>PV2.mode) and
        (PV1.[HT, HM, STM] <= PV2.[HT, HM, STM])
   return TRUE;
 else return FALSE;
}
```

Figure 5: Comparing Privilege Vector guarantees.

([WRLK, *, *, *] where * is a wildcard), whereas a new replica starts with the lowest PV ([RD, ∞, ∞, ∞]). Associated with each replica of a file or file block 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 replica remembers, for each neighbor N, 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 a replica's PV state is proportional to its fanout and not to the total number of replicas.

The bulk of the CM's functionality consists of the pull and push 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 PV says it is read-only), the CM performs a pull operation. The pull operation obtains the consistency guarantees represented by a PV from those neighbors whose PVin is not strong enough to cover the PV. by issuing `get` messages to them in parallel. In response to a `get` message, those neighbors recursively pull the PV from their other neighbors, and recompute their own PV to be compatible with the new PV. Finally they reply with the new PV and any pending updates via `put` messages. 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 guarantee.

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, the local CM sends them via `put` messages to its neighbors if their divergence control requirements demand it.

To ensure fairness to client requests, concurrent pull operations for a particular file on a node are performed in FIFO order. As a replica issues `get` requests in parallel to neighbors in the hierarchy, pull latency grows logarithmically with the total number of replicas provided the hierarchy is kept dense by Swarm.

Enforcing Replica Divergence Bounds: To enforce a hard time bound (HT), a replica R issues a pull to neighbors with a write mode in their PV 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.

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 leases expire, which lets it recover from a lease-holding node 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.

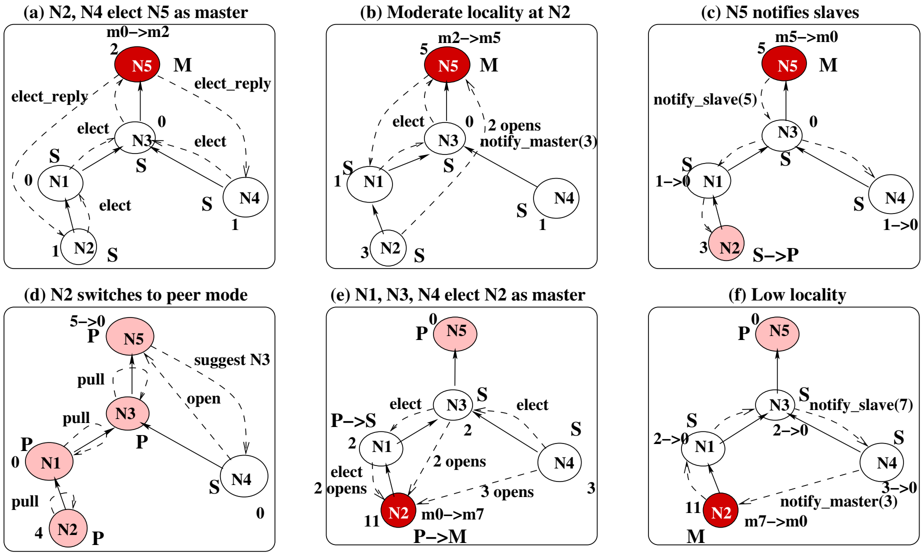
Update Propagation: Swarm propagates updates (modified file blocks or operational updates) via put messages. Each Swarm server stores operational updates in FIFO order in a persistent *update log*. For the purpose of propagation, each update from a client session is tagged by its origin replica and a 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 originating 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. Servers 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 Unix-style write mode (WR) sessions. 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 [4] and uses version vectors.

5 Contention-Aware Caching

In this section, we describe Swarm's contention-aware caching scheme. Swarm's basic replication algorithm works as follows. Every Swarm server tracks how often it receives a request to access a particular piece of data since it last had its access revoked. Until it receives an adequate number, it attempts to acquire the data from a *master* node. If no master exists, it initiates a master election process. Once a server accesses a particular piece of data sufficiently often without interruptione, it attempts to switch into peer-to-peer


(a) N2, N4 elect N5 as master

(b) Moderate locality at N2

(c) N5 notifies slaves

(d) N2 switches to peer mode

(e) N1, N3, N4 elect N2 as master

(f) Low locality

Figure 6: Contention-aware Caching of file F2 of Figure 3 with hysteresis set to (low=3, high=5). In each figure, the master mode replica is darkly shaded, peers are lightly shaded, and slaves are unshaded.

sharing mode and cache a replica of the data locally. The heart of Swarm's contention-aware caching scheme is the way it handles this attempt to switch into peer-to-peer caching mode. If Swarm determines that there is adequate access locality, it allows the server to become a full-fledged peer sharer, otherwise it tells the server to remain in master-slave mode. As a result, data will be widely replicated when contention is low, but serviced via RPC when contention is high. In the remainder of this section, we describe how Swarm measures contention, elects a suitable master replica to which client requests should be forwarded during periods of high contention, and reverts back to peer caching mode when contention subsides.

Tracking Contention: Each replica monitors the number of access requests (i.e., session `open()` requests) it satisfies locally since the last time its access privilege was revoked. This reuse count provides an indication of how useful it would be to replicate the data locally. For instance, if the data is read-mostly and the local server services many client read requests between successive remote writes, the count will be high, indicating that replication is potentially useful. In contrast, if clients issue strictly interleaving write sessions at multiple replicas of a file employing close-to-open consistency, their reuse counts will all be zero, indicating that replication would not be useful.

Replicas piggyback their reuse counts on consistency messages exchanged with their neigh-

```
check_locality(): /* invoked before pull */
  if reuse $<$ soft_threshold,
    return elect_master();
  else if reuse $<$ hard_threshold {
    if i am slave,
      send notify_master(reuse) to master;
  }
  else master = none; /* PEER */
  return master;

elect_master():
  if master $\neq$ none,
    return master;
  find replica N (parent, self, children)
    where N.reuse is highest;
  if N $\neq$ self,
    send elect_master() msg to N;
    await reply; /* and become SLAVE */
  else master = self; /* MASTER */
  return master;
```

Figure 7: Pseudocode for Contention-aware Swarm.

bors in the replica hierarchy (e.g., get and put operations). They use this information to keep track of which node has the highest local reuse count (i.e., themselves or one of their neighbors).

Electing a master replica: Recall that Swarm allows clients to specify soft and hard caching thresholds when they open a session on a replica. Figure 6 illustrates the algorithm's operation in the context of accesses to file F2 of Figure 3, when employing soft and hard caching thresholds of 3 and 5. When a server needs to pull data in response to a local request, it first checks its local reuse count for the data (see Figure 7). All replicas start with a zero reuse count. If the count is below the soft caching threshold, the server must locate a *master (M)* replica. In our example, in response to receiving open requests, nodes N2 and N4 initiate elections (see Figure 6a). To locate a master, a server sends an elect_leader request up the replica hierarchy to find an already elected master replica or find a replica that has accumulated a high reuse count. The requesting server switches to *slave (S) mode*, i.e., creates a session at the master and services the local reads and writes via the master. Both the slave and master replica increment their local reuse counts. Figure 6(a) illustrates the situation after nodes N2 and N4 (reuse count of 1) have each initiated a single master-slave sharing session with master node N5 (reuse count 2).

When a slave's reuse count reaches the soft replication threshold, e.g., as illustrated in Figure 6(b) after node N2 has received two more local requests, the slave sends a notify_master message to the master, along with the current reuse count. The master compares the slave's reuse count with its own, and if it determines that the slave is a major contributor to the total number of accesses to the data, it sends a message to the slave to inform it that it can switch to peer-to-peer mode and cache the data locally (see Figure 6(c)). When N4 subsequently issues an open request to N5, N5's use count has dropped to 0, so it suggests N4 contact N3 (see Figure 6(d)).

Should a slave's reuse count reach the hard replication threshold, the slave unilaterally switches to *peer* (P) caching mode). The hard threshold is used to recover from non-responsive (e.g., failed) master nodes.

Continuing our example, suppose N1, N3 and N4 issue uniformly interleaved open requests while N2 continues to handle a large volume of client traffic. In this case, illustrated in Figure 6(e), node N2 is elected as the new master due to its recent high activity.

When there is high contention for a piece of data, it is best to disable replication and stick with a single master until contention drops. This scenario is illustrated in Figure 6(f), where nodes N1, N3, and N4 have issued 2, 2, and 3 (respectively) close-to-open write session requests to master node M2. When node N4's notify_master message arrives at N2 with a reuse count of 3, node N2 determines that N4 is not responsible for a majority of the 7 total accesses, and denies the request to upgrade to peer-to-peer mode. Instead, it sends a `notify_slave` message to all slaves in the replica hierarchy, along with its reuse count (7). If a slave sees that it contributed to a major portion of the master's reuse count (via session forwarding), it switches to peer mode without waiting for its count to reach the hard threshold. Otherwise it assumes that there is contention at the master, resets its reuse count, and stays in slave mode. Note that ideally one of nodes N1, N3, and N4 would become the master, since N2 is no longer accessing the data, but our algorithm does not achieve this optimal behavior. That said, by simply disabling replication during contention, our mechanism achieves good performance. As an optimization, once a master has issued a `notify_slave` message, indicating that the data is experiencing high contention, it resends `notify_slave` messages every time it recrosses the soft threshold. This optimization keeps slaves from needlessly pestering the master while contention remains, while at the same time allowing fast master migration should any one node become the dominant accesser of the data.

Discussion: Our contention-aware caching algorithm has several beneficial properties. When replicas handle sufficient local requests between successive remote pulls, e.g., when data is infrequently written, data is aggressively cached after a small period in master-slave mode. When there is high contention but some replicas sees significantly more accesses than others (moderate locality), the most frequently accessed replica is elected master, and other client accesses are redirected to it. Since master election stops at a replica with reuse count higher than a threshold, there can be multiple master mode replicas active at a time, in which case they are all peers to one another. When client accesses are interleaved "randomly", e.g., due to widespread contention (poor locality), one of the replicas is elected master and data is served in master-slave mode until contention eases. Once a node becomes master, it is not demoted until remote usage indicates that contention has eased sufficiently for slave reuse counts to exceed a threshold in the face of frequent `notify_slave` operations. This hysteresis prevents frequent oscillation between master-slave and peer-to-peer replication modes. One known imperfection of our caching algorithm that it is not guaranteed to elect the optimal master during periods of high contention.

6 Evaluation

We now evaluate Swarm's replication mechanisms in a dynamic wide area environment and under varying data contention. In Section 6.1, we evaluate the network economy and failure-resilience of Swarm's replica hierarchy formation using a large-scale file caching workload involving a hundreds of peer nodes continuously joining and leaving the network (node churn). Our evaluation reveals that though Swarm consumes network bandwidth when a number of Swarm nodes discover each other initially, its WAN usage quickly subsides and file download latency improves due to efficient replica networking and node membership management.

In Section 6.2, we evaluate the effectiveness of Swarm's contention-awareness under strongly consistent page caching across WAN links. With a synthetic enterprise object proxy caching workload modeled after the TPC-A benchmark, we show that contention-aware caching outperforms aggressive caching under high locality, high contention as well as dynamic shifts in working sets of objects at caching sites.

For all experiments, we used the Emulab Network Testbed [20] to emulate WAN topologies among clustered PCs. The PCs have 850MHz Pentium-III CPUs with 512MB of RAM and run FreeBSD 4.9 or Redhat 9. The Swarm replica fanout was configured to a low value of 4 to induce hierarchies 4 to 5 levels deep.

6.1 Aggressive File Caching under Node Churn

To evaluate Swarm's replication under node churn, we emulate a large file sharing network where a number of wide area peer users continually join and leave the sharing network, browsing files at their personal workstations via local Swarm servers. Each peer comes online, repeatedly looks up files in a shared index and downloads previously unaccessed files for a while by accessing them at its local Swarm server, and abruptly goes offline, killing the server as well. To evaluate how Swarm's privilege-leasing performs under churn, we employ close-to-open consistency for shared files without updating them to study replica formation performance in isolation. This forces Swarm to maintain leases, forcing a replica to make sure it has the latest contents and valid lease, re-electing a parent if needed, before it serves the file.

We emulate the network topology shown in Figure 8. Machines owned by users are clustered in campuses spread over multiple cities across a wide area network. Campuses in the same city are 10Mbps and 10ms RTT apart. Campuses in neighboring cities are connected by a backbone router, and are 5Mbps and 50ms apart. Campuses separated by the WAN are 5Mbps and 150ms apart and need to communicate across multiple backbone routers (two in our topology). Each user node runs a file sharing agent and a Swarm server, which are started and stopped together. To conduct large-scale experiments on limited physical nodes, we emulate 10 user nodes belonging to a campus on a single physical machine. Thus, we emulate 240 user nodes on a total of 24 physical machines. The CPU, memory, and the disk were never saturated on any of the machines during our experiment.

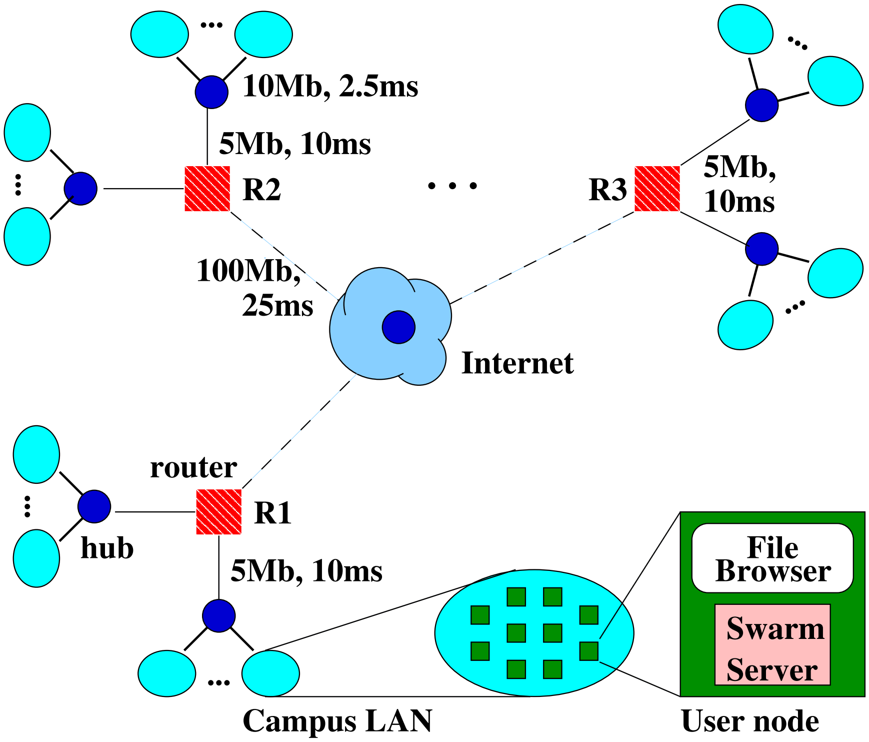


Figure 8: Emulated network topology for the large-scale file sharing experiment. Each oval denotes a ‘campus LAN’ with ten user nodes, each running a file browser and Swarm server. The labels denote the bandwidth (bits/sec) and oneway delay of network links (from node to hub).

We designated 10% of the nodes as 'home' nodes (i.e., custodians) that create a total of 1000 files and make them available for sharing by adding their keys (i.e., attribute-value pairs) to a Swarm-based shared BerkeleyDB [18] index. These nodes are configured never to go down, because our Swarm prototype's SWID lookup mechanism cannot recover from custodian failures. Once every 1-3 seconds, each node looks up the shared index for files using a uniformly random key and downloads via Swarm those not available locally. We choose 50KB files as they are large enough to stress network link bandwidth, but small enough to expose the overheads of Swarm's replica hierarchy formation. We initially bring up the home nodes and then the other nodes in random order, one every 2 seconds. We run them for a warm up period (10 minutes) to let Swarm build replica hierarchies for various files. We then churn the non-home nodes for a period of 30 minutes, followed by a quiet (i.e., no churn) period of 10 minutes before shutting down all the nodes. Peer nodes have an exponentially distributed *lifetime* (i.e., time spent online). We simulate median node lifetimes of 30 seconds, 1 minute, and 5 minutes. We start a new node each time one is killed to maintain the total number of nodes constant during the experiment. A similar churn model was described by Liben-Nowell et.al. [10] and employed by Bamboo DHT [16]. These short node lifetimes expose the sensitivity of Swarm's replica networking and consistency mechanisms to the disruption caused by high node churn rates (ranging from an average of 5 node deaths/sec to a node dying every 2 seconds).

We compare three configurations of Swarm with different replica networking schemes. In the first configuration denoted *random hierarchies*, we disable Swarm's proximity-awareness and its associated distance estimation mechanism, so that replicas randomly connect to others willing to serve them without regard for network distances. In the second configuration denoted *eager download*, we enable proximity-aware replication (indicated as *WAN-aware* in the figure), but make a new replica eagerly start downloading file contents as soon as it connects to a parent, and not disrupt its parent association until it finds *and* connects to a nearer parent replica in the background. This causes the file download to start from the first accessible parent replica. In the third configuration (*deferred download*), we make a new replica search for a nearer parent a little more aggressively before it starts downloading file contents as follows: so long as a replica does not have valid file contents, it proactively disconnects from its old parent as soon as it learns of a nearer parent candidate. The eager download scheme avoids disrupting an established replica hierarchy. The intuition behind the deferred scheme is that spending a little more time initially to find a nearer parent allows better network utilization.

Results: Figure 9 shows the mean latency observed for new file accesses at all nodes as a timeline for the three configurations. For the *WAN-aware* schemes, the file access latencies are initially high as a large number of nodes come online for the first time, learn about each other, and find relative network distances by pings. However, the average access latency quickly drops to about 400ms as the network traffic subsides and nodes learn about

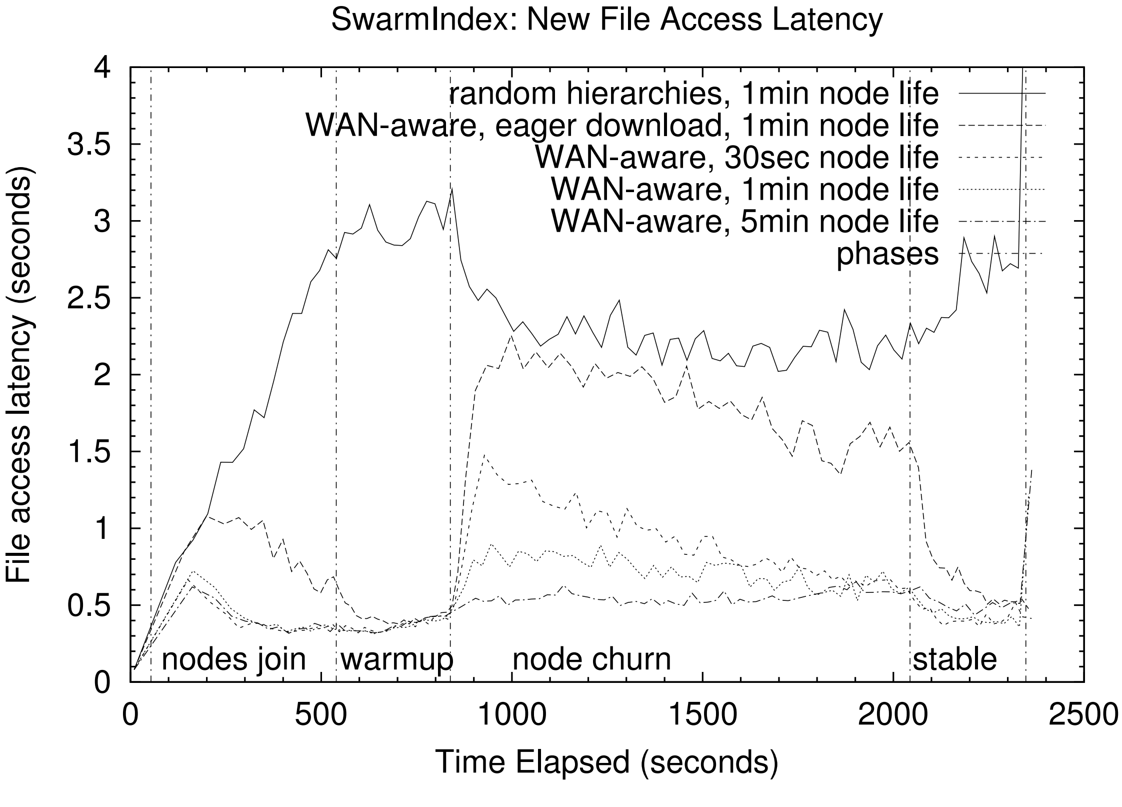


Figure 9: New file access latencies in a Swarm-based peer file sharing network of 240 nodes under node churn.

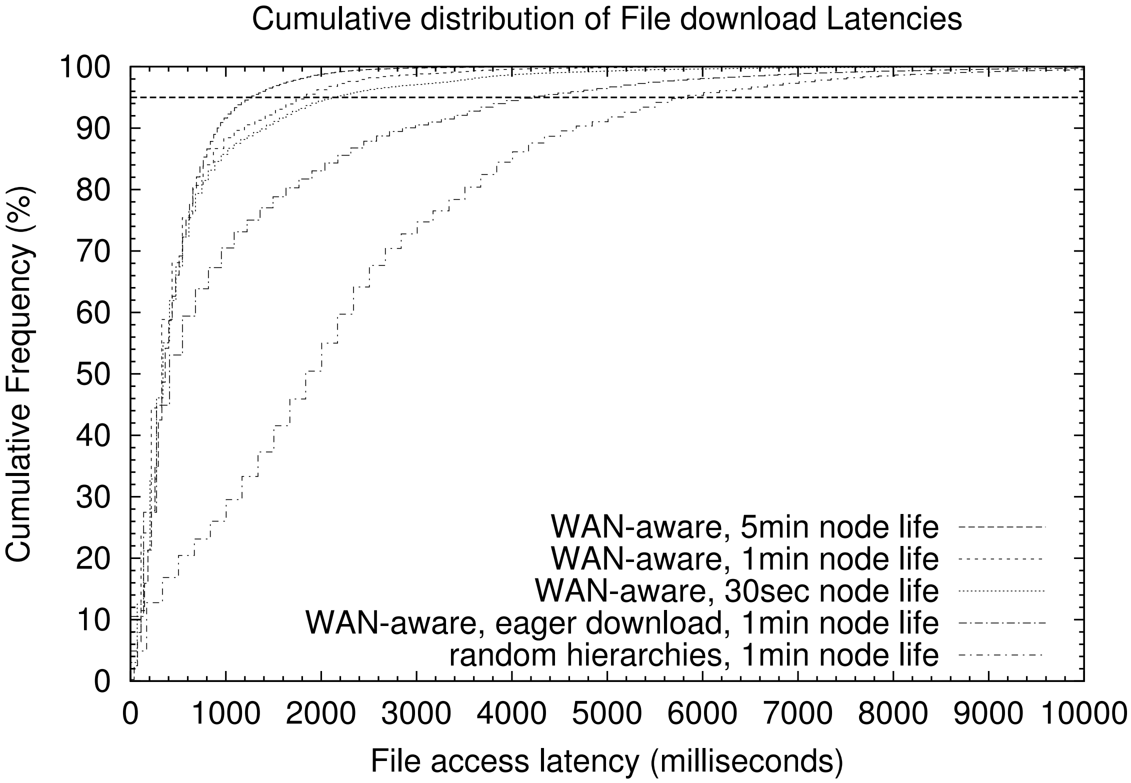


Figure 10: Cumulative distribution of new file access latencies in various Swarm configurations.

nearby nodes and replicas. On average, replicas tried about 3 to 5 parent candidates before downloading a file. Since a ping across routers takes 150ms and a download across 10Mbps campus links takes 40ms, this indicates that files are almost always downloaded from within a campus.

In the *random hierarchies* scheme, file access latencies do not improve with time. Though the WAN-aware schemes are affected by node churn, their network economy property still holds, and results in 6 times better latencies than with the random scheme. The eager download scheme performs 2.5 times worse than the deferred scheme, as a new replica often initiates file download from a far away replica before it finds and connects to a nearby parent. The latter scheme utilizes the network much more efficiently resulting in better latencies that approach those in the stable network phase even with a node joining and leaving the network every 2 seconds (i.e., median node life of 5 minutes). Figure 10 shows the cumulative distribution of download latencies.

We also plotted the incoming bandwidth consumed on a router (R1)'s high-latency WAN link by Swarm traffic in various configurations. The graph was similar to the access latency graph of Figure 9 and hence is omitted. Random scheme incurred 750KB/sec during churn while WAN-aware Swarm incurred 70KB/sec without churn and a peak of 150KB/sec (one-fifth of random) with churn.

To evaluate the overhead imposed by Swarm's background reconnections in the replica hierarchy, we disabled them but did not see much improvement in performance. This shows that their overhead is not significant under high churn at this scale. Failed file accesses were less than 0.5%, mainly due to the error returned to file sharing agents' last file request when their local Swarm servers had already died. Finally, less than 0.5% of the messages received by any node (under all churn levels) were root probes, indicating that our cycle detection scheme has negligible overhead.

6.2 Contention-aware Caching

To evaluate how Swarm's contention-aware caching technique supports strongly-consistent wide-area caching of enterprise objects (such as sales and customer records) in applications such as online auctions, we designed a simple synthetic proxy-based object-caching workload modeled after the TPC-A benchmark.

As we argued in Section 5, contention-aware caching does not degrade availability below what Swarm's peer caching provides. When a Swarm-based proxy server fails, clients using

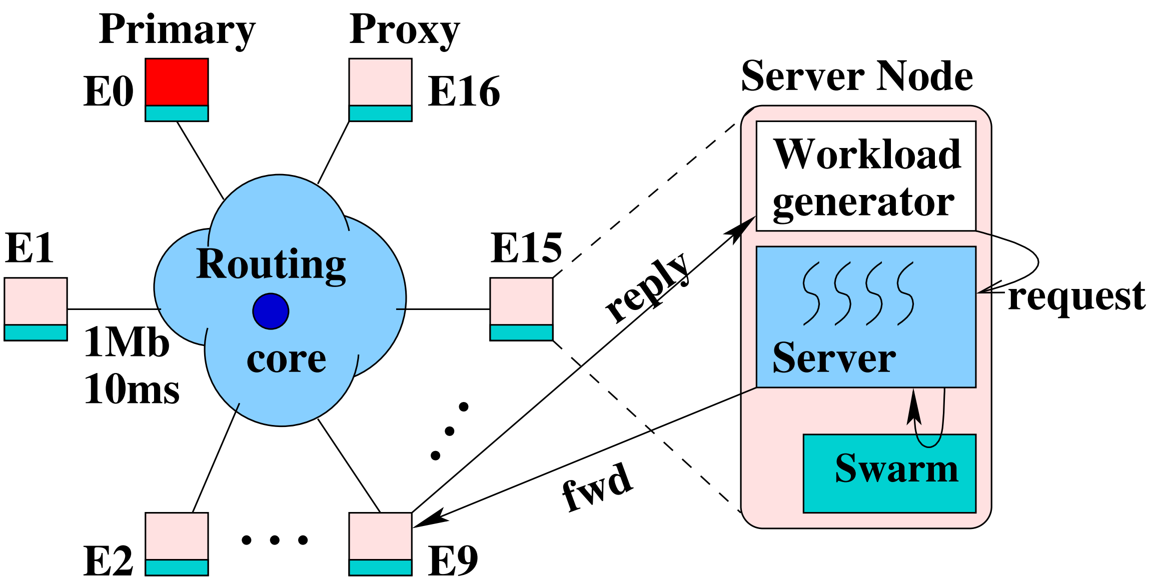


Figure 11: Network Topology emulated in SwarmProxy experiments.

it can be simply redirected to use other proxies. Swarm's leasing and replica hierarchy repair mechanisms enable the other proxies to quickly recover from its failure.

Architecture: Our workload models that of a web-based three-tier enterprise service whose middle-tier server accepts service requests from front-end web servers and operates on data stored in a back-end object database. Our Swarm-based version deploys wide area service proxies (called *SwarmProxies*) to serve regional clients by caching enterprise objects locally on demand. Unlike existing clustered architectures for web-based enterprise services[5], our web servers run next to the service proxies deployed at various geographic locations. As all the enterprise servers (primary and proxies) accessing data via Swarm they are functionally identical. Figure 11 shows the architecture we emulate in our experiments. It consists of a primary server site E0 that hosts the object database in Swarm and 16 other sites (E1-E16) connected to a common Internet router core by 1Mbps, 10ms delay links, with 40ms roundtrip between them. We run Redhat 9 on all nodes.

Workload: We model the work of a heavily loaded multi-threaded web server by running four full-speed workload threads (referred to as *clients*) at each of the nodes E1-E16. They repeatedly issue queries and updates on enterprise objects as RPCs to a local service proxy process if available, or to the primary enterprise server at node E0 otherwise. Our synthetic enterprise database consists of two components: an index structure and a collection of 256 enterprise objects (each 4KB), all stored in a single Swarm file managed as a persistent page heap (with a page size of 4KB). The index structure is a page-based B-tree that maps object IDs to offsets within the heap where objects are stored. We deliberately employed a relatively small object space to induce a reasonable degree of sharing and contention during a short period of experimental time.

Processing a request at a SwarmProxy involves walking the B-tree index in RDLK mode to find the requested object's heap offset from its OID, and performing the operation after locking the object's page in an appropriate mode. Thus, the index pages are read-only replicated whereas the object pages incur contention due to read-write replication. The SwarmProxies employ contention-aware caching for object pages. Thus, a proxy can sometimes forward requests to a remote proxy suggested by Swarm, which finally replies to the client.

Performance Metrics: We use two metrics to evaluate SwarmProxy performance: the aggregate service throughput (client requests processed per second at all web servers combined), and the latency observed by front-end web servers for their requests. We expect that as proxies get deployed near web servers, the aggregate service throughput scales better with contention-aware caching than with aggressive caching. Ideally, we expect less contended objects to incur near-local latency, and the latency of heavily contended objects not to degrade beyond that of a WAN RPC. To evaluate Swarm's performance under heavy write contention, we model a high (50%) proportion of writes.

Simulating Contention: We vary the degree of contention as follows. Associated with each of the 16 web servers are 16 "local" objects (e.g., clients on E1 treat objects 1-16 as their local objects etc.). When a client randomly selects an object on which to operate, it first decides whether to select a local object or a "random" object from the entire set. We vary the likelihood of selecting a local object from 0%, in which case the client selects any of the 256 objects with uniform probability, to 100%, in which case the client selects one of its node's 16 local objects with uniform probability. In essence, the 100% case represents a partitioned object space with maximal throughput because there is no sharing, while the 0% case represents a scenario where there is no access locality.

Experiments: We evaluate SwarmProxy performance via three experiments run in sequence. Our first experiment evaluates the effect of adding wide area proxies on the overall service performance. We run the primary enterprise server on node E0 and deploy web servers running the workload described above on each of the 16 nodes (E1-E16). Thus initially, all web servers invoke the primary server via RPCs across WAN links, emulating the traditional client-server organization with no caching. Subsequently, we start a SwarmProxy (and associated Swarm server) at a new node (E1-E16) every 50 seconds, and redirect the "web server" on that node to use its local proxy. As we add proxies, Swarm caches objects near where they are most often accessed, at the cost of potentially increasing the coherence traffic needed to keep individual objects strongly consistent.

Our second experiment evaluates how Swarm automatically adapts to dynamic shifts of data locality in client requests while fixing the amount of locality at 40%. After we deploy SwarmProxies at all web servers, each client shifts its notion of what objects are "local" to be those of its next cyclical neighbor (Clients on E1 treat objects 16-31 as local etc.). We run this scenario denoted as "locality shift" for 100 seconds.

Our third experiment evaluates how Swarm performs when many geographically widespread clients contend for a very few objects, as when bidding for a popular auction item. After experiment 2 ends, clients on nodes E9-E16 treat objects 1-16 as their "local" objects, which introduces very heavy contention for those 16 objects. We run this scenario denoted as "contention" for 100 seconds.

We run the above experiments in two modes: *eager replication mode*, where Swarm servers always cache objects locally on access, and *contention-aware replication mode*, where Swarm adapts between peer replication and master-slave modes to prevent thrashing. For the contention-aware case, we set the database file's soft and hard caching thresholds (defined in Section 2) to 6 and 9.

Results: Figures 12 and 13 show how aggregate throughput varies as we add SwarmProxies in experiment 1 under different degrees of locality (0%-100%) when strong consistency is enforced. Vertical bars denote the addition of a proxy near a web server. To determine the speedup achieved by adding WAN proxies, we measured the maximum throughput that a single saturated enterprise server can deliver. For this, we ran a single web server's workload with its four full-speed threads when it is colocated with the primary server on node E0. As the line labeled "local" in the graphs indicates, it is about 1000 ops/second. The line labeled "rpc" denotes the aggregate throughput achieved when all the 16 web servers run the same workload on the single primary server across the WAN links without proxies. It is 480 ops/sec, well below what a single server can deliver.

At 100% (i.e., perfect) locality which indicates that clients always access objects "local" to their region, the addition of proxies causes near-linear speedup as expected, with both eager and contention-aware replication. Thus, Swarm automatically partitions the data to exploit perfect locality if it exists. Also with both replication modes, the higher the degree of locality, the higher the throughput at the front-end web servers.

With eager replication, aggregate throughput quickly levels off as more proxies contend for the same objects. Even at 95% locality, throughput never exceeds 2500 ops/sec (i.e., a speedup of 15.6% with 16 proxies), as cache misses incur a heavy penalty by disrupting the locality at frequent-use sites as well. With contention-aware replication, clients initially forward all requests to the primary server on E0, but when locality exceeds 40%, Swarm-

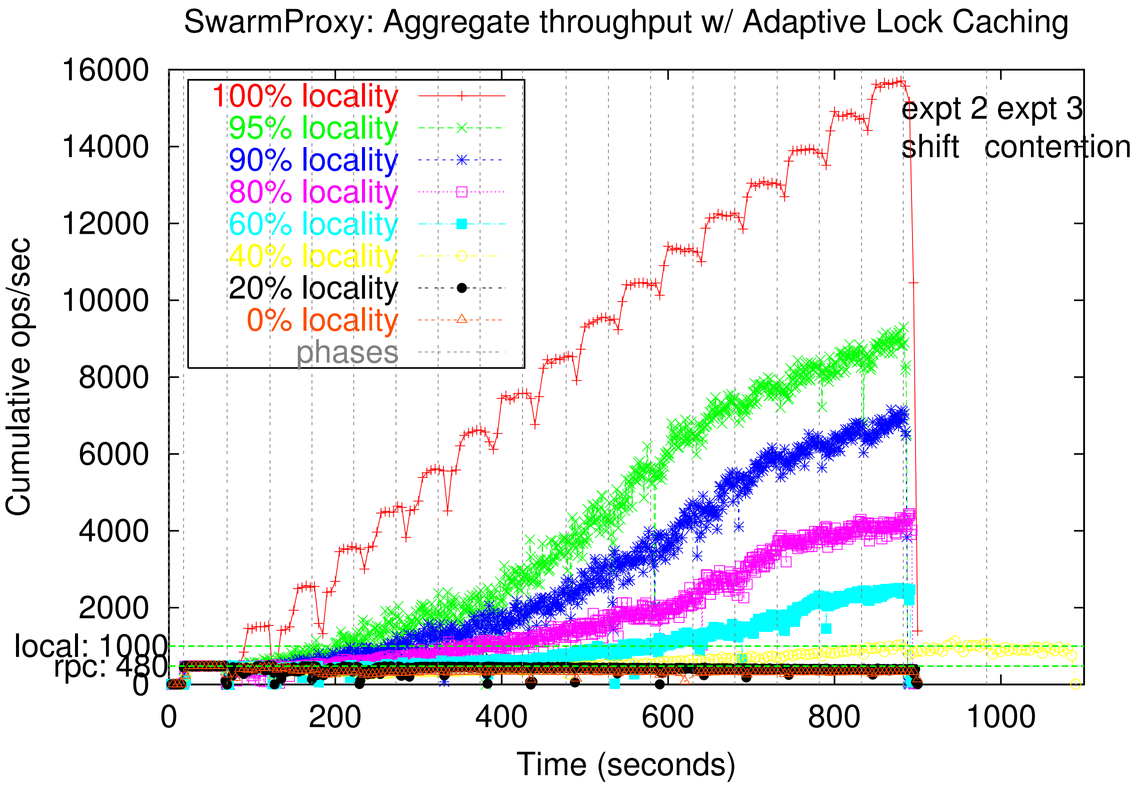


Figure 12: SwarmProxy aggregate throughput (ops/sec) with contention-aware replication.

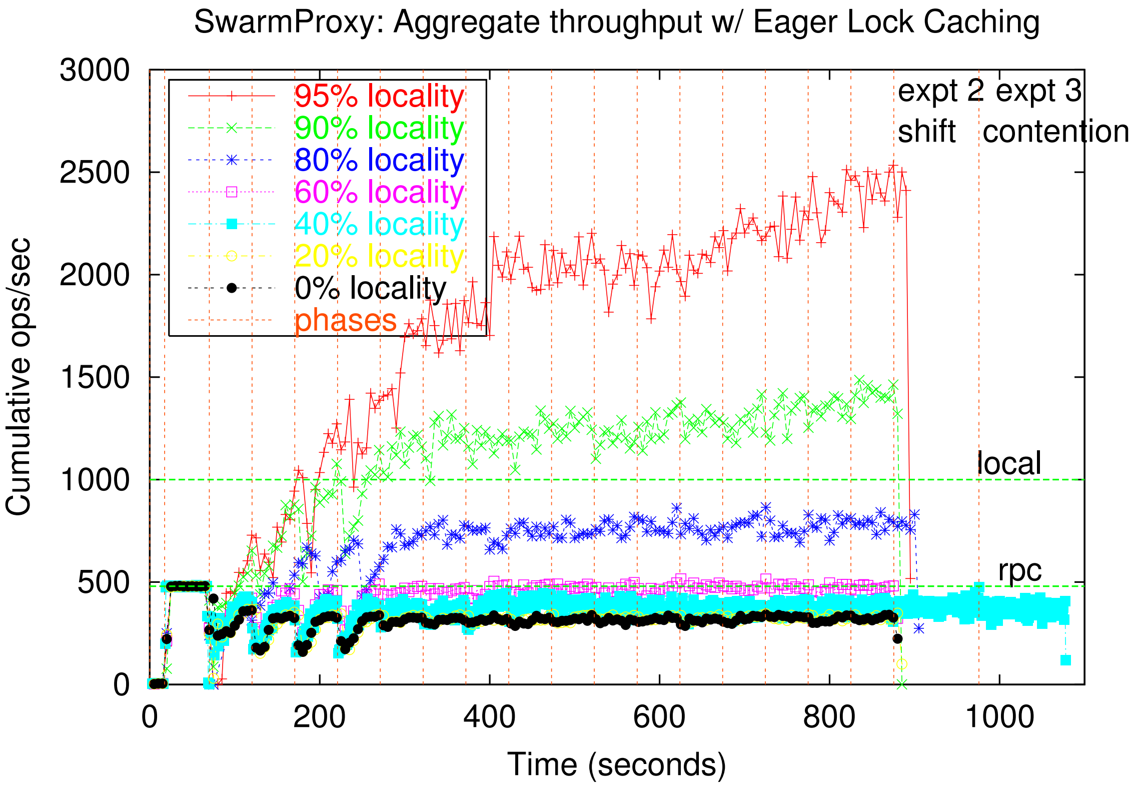


Figure 13: SwarmProxy aggregate throughput (ops/sec) with eager replication.

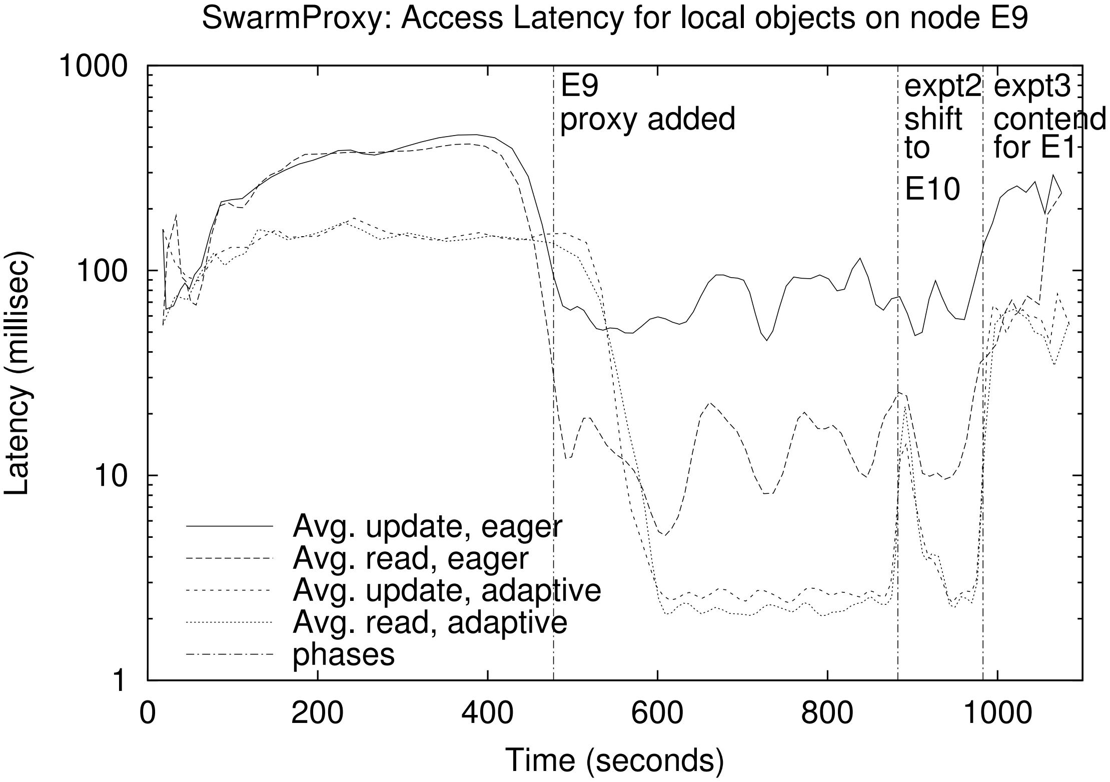


Figure 14: SwarmProxy latencies for local objects.

Proxies quickly cache their "local" objects soon after they are spawned. Under these circumstances, nodes will use RPCs to access "remote" objects, rather than replicating them, which eliminates thrashing and allows throughput to continue to scale as proxies are added. With high (95%) locality, the contention-aware protocol can support almost 9000 ops/sec in the fully proxy-based system (a speedup of 56% and a 3.6x improvement over the eager protocol). Further improvement was not achieved because our contention-aware algorithm incurs some communication overhead to track locality, which can be optimized further.

Figures 14 shows the distribution of access latencies for "local" objects on node E9 throughout the experiment. Even with modest (40%) locality, the contention-aware protocol reduces the access latency to "local" objects from 100msecs to under 5msecs once a local proxy is spawned. In contrast, when using eager replication, the average access latency hovers around 100msecs due to frequent lock shuttling. Figure 15 shows that the contention-aware mode outperforms the eager mode for access to "non-local" objects as well, despite never caching these objects locally, by eliminating extra coherence traffic.

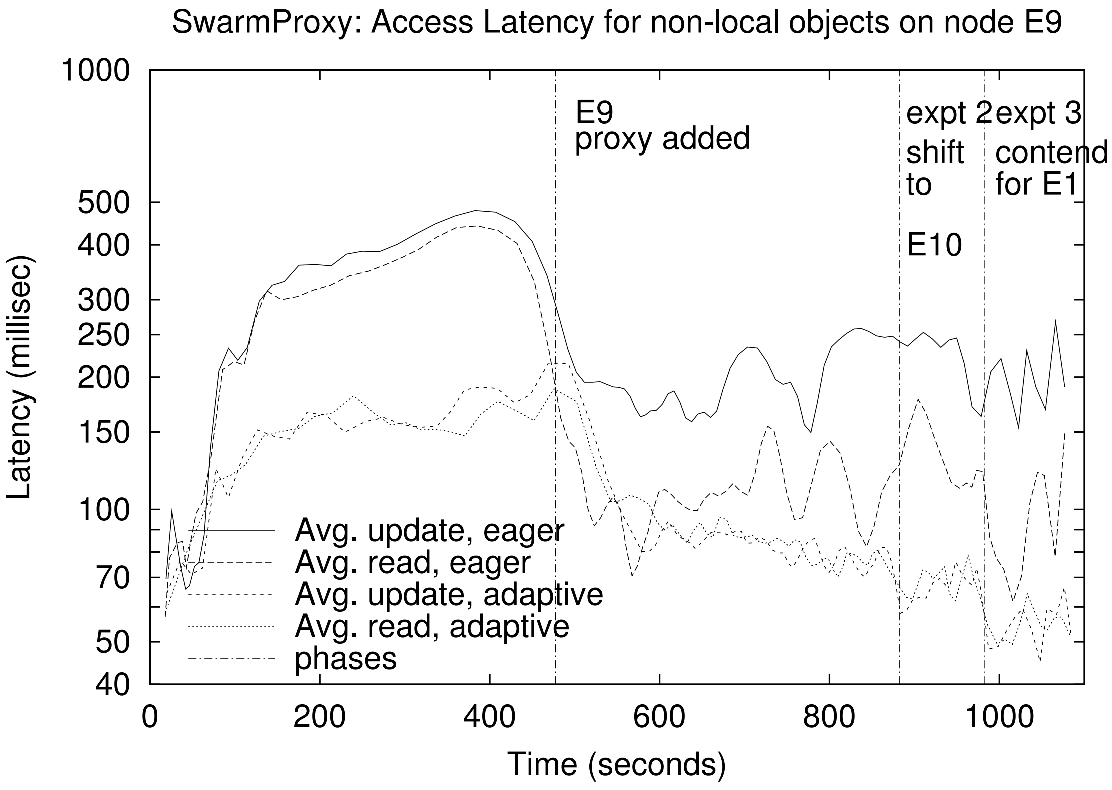


Figure 15: SwarmProxy latencies for non-local objects.

When we have each node shift the set of objects that most interest it, the phase denoted “expt2” in the graphs, the contention-aware protocol migrates each object to the node that now accesses it most often after about 10 seconds, as seen in Figure 14. Performance of the eager caching does not change, nor does the average access latency of “non-local” objects.

Finally, when we induce extremely heavy contention for a small number of objects, the phase denoted “expt3” in the graphs, the contention-aware protocol almost immediately picks a single replica to cache the data and shifts into RPC mode. By doing so, it is able to serve even heavily accessed objects in under 100msecs at node E9. In contrast, we found that the eager replication protocol often requires over 500msecs to service a request and only rarely sees sub-100msec latency.

In summary, Swarm’s contention-aware caching provides better performance than aggressive caching and client-server RPCs at all degrees of data locality.

7 Related Work

Numerous solutions exist to manage caching and replication in non-uniform or wide-area networks [4, 7, 14, 17]. Many support weak consistency semantics to improve availability and caching performance, as does Swarm. Swarm adopts existing techniques for P2P data location [15], update propagation, ordering, version maintenance [4], and fault-tolerance [8].

Database and object systems have long recognized the performance tradeoffs between function-shipping and data-shipping [2, 12, 13]. Many provide both paradigms, but require the choice to be made at application design time. In contrast, Swarm allows applications to make the choice dynamically. The Sprite [11] file system disables client caching when a file is write-shared. Swarm chooses the frequently accessed copy as the master. The Munin DSM system [3] freezes a locking privilege for a minimum time (hundreds of milliseconds) before allowing it to migrate away, which works well in a LAN. Munin still allows data shuttling, which quickly offsets the speedup achieved by hysteresis in WANs.

Pangaea [17] provides aggressive replication in a peer file system. It organizes replicas as a graph and floods updates among its edges to provide eventual consistency. Swarm employs a cycle-free topology and recursive messaging, and provides much stronger semantics and comparable network economy and failure resilience compared to Pangaea.

A recent study of node churn in DHTs [16] recommends that systems not react to a failure by immediately reorganizing their overlay topology, as this will likely cause a positive feedback cycle of more failures due to congestion. In contrast, Swarm uses TCP to detect node failures and react rapidly, but employs much longer timeouts with busy neighbors to avoid positive feedback cycles.

8 Conclusions

In this paper, we presented and evaluated the design of a middleware system that supports the wide-area data caching needs of diverse distributed services. Swarm builds failure-resilient dynamic replica hierarchies to manage large numbers of widely dispersed replicas. Swarm supports a *contention-aware caching* mechanism that typically caches data very aggressively, but restricts replication when it observes high contention. Swarm supports a *proximity-aware caching* mechanism wherein Swarm continually monitors the network quality between clients and replicas and reorganizes the replica hierarchy to minimize the use of slow links. Swarm allows applications to tune consistency to their availability needs. Finally, Swarm is resilient to network and node failures, and handles node churn gracefully.

Our evaluation demonstrated that Swarm's replication mechanisms are network-efficient, contention-aware, and failure-resilient. Proximity-aware caching improves P2P file access latency and reduces WAN bandwidth consumed on a 240-node system by 80% compared to random replication. Even with a high node churn of 5 node deaths/sec, performance degrades only 20% as much using Swarm's replication mechanisms compared to a non-adaptive mechanism. Finally, we found that Swarm's contention-aware caching mechanism outperformed static caching mechanisms at all levels of contention for an enterprise service workload.

References

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

[2] K. Bohrer. Architecture of the san francisco frameworks. *IBM Systems Journal*, 37(2), Feb. 1998.

[3] J. Carter. Design of the munin distributed shared memory system. *Journal of Parallel and Distributed Computing*, 29(2):219–227, Sept. 1995.

[4] 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.

[5] A. Fox, S. Gribble, Y. Chawathe, E. Brewer, and P. Gauthier. Cluster-based scalable network services. In *Proceedings of the 16th Symposium on Operating Systems Principles*, Oct. 1997.

[6] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. Idmaps: A global internet host distance estimation service. *IEEE/ACM Trans. on Networking (TON)*, 9(5):525–540, Oct. 2001.

[7] J.Kubiatowicz et al. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th Symposium on Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[8] P. Keleher. Decentralized replicated-object protocols. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, Apr. 1999.

[9] 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.

[10] D. Liben-Nowell, H. Balakrishnan, and D. Karger. Analysis of the evolution of peer-to-peer systems. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*, July 2002.

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

[12] Object Management Group. The Common Object Request Broker: Architecture and Specification, 1996.

[13] Oracle Corp. *Oracle 7 Server Distributed Systems Manual, Vol. 2*, 1996.

[14] D. H. Ratner. ROAM: A scalable replication system for mobile and distributed computing. Technical Report 970044, University of California, Los Angeles, 31, 1997.

[15] 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.

[16] S. Rhea and D. Geels and T. Roscoe and J. Kubiatowicz. Handling Churn in a DHT. In *Proceedings of the USENIX 2004 Annual Technical Conference*, June 2004.

[17] 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.

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

[19] 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.

[20] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. 5th Symposium on Operating System Design and Implementation*, Boston, MA, Dec. 2002.

[21] 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.