Building Data Structures on Untrusted Peer-to-Peer Storage with Per-participant Logs

Benjie Chen Thomer M. Gil Athicha Muthitacharoen Robert Morris
MIT Laboratory for Computer Science
200 Technology Square, Cambridge, MA 02139
$\{$*benjie,thomer,athicha,rtm*$\}$*@lcs.mit.edu*

Abstract

L^* is a technique for building multi-user distributed data structures out of untrusted peer-to-peer distributed hash tables (DHTs). L^* uses multiple logs, one log per participant, to store changes to the data structure. Each participant finds data by consulting all logs, but performs modifications by appending only to its own log. This decentralized structure allows L^* to maintain meta-data consistency without locking and to isolate users' changes from each other, an appropriate arrangement for unreliable users.

Applications use L^* to maintain consistent data structures. L^* interleaves multiple logs deterministically so that decentralized clients can agree on the order of completed operations, even if those operations were issued concurrently. When the data structure is quiescent, L^* guarantees that clients agree on the state of the data structure. L^* optionally provides mutual exclusion for applications that need to ensure atomicity for multi-step operations. The Ivy file system, built on top of L^*, demonstrates that L^*'s consistency guarantees are useful and can be used and implemented efficiently.

Regular submission. The first three authors are students.
Please consider paper for brief announcement as well.

1 Introduction

Recent peer-to-peer distributed hash tables (DHTs) [1, 9, 11, 4, 16] promise to support a new approach to certain kinds of network storage applications. These DHTs provide a simple API allowing read and write of key/value pairs (often called blocks). The DHT typically takes care of finding a network host to store each key/value pair; replicating data for availability; and checking that retrieved blocks have not been tampered with. The DHT interface is fairly low level, much like the sector read/write interface of a disk drive. Thus, applications often build complex data structures on top of DHTs, with blocks containing pointers (keys) to other blocks. For example, CFS [1] builds a file system on top of a DHT, storing each file and directory in a separate block; a directory contains a list of DHT keys referring to the files in the directory.

While DHTs defend the availability and integrity of individual blocks against unreliable and malicious DHT nodes and clients, an application that uses a DHT typically has additional consistency invariants that it would like to maintain on the data structure it stores in the DHT. For example, a client crash during a file rename in a DHT-based file system should not leave the file system in an incorrect state. Because clients in a DHT-based application typically manipulate a shared data structure independently (i.e. without sending operations to a single server or server cluster), an application with concurrent clients also faces the challenge of providing consistency without direct use of serialization. Additionally, peer-to-peer systems are often used in situations where clients do not fully trust each other; thus another problem is how to defend against clients who maliciously damage the shared data structure. Finally, DHTs typically replicate data in such a way that multiple partitions may have a complete copy of the data structure if a network outage occurs; thus applications using DHTs may experience conflicting updates in different partitions.

This paper presents L^*, a set of techniques for maintaining consistent data structures in DHTs. L^* represents the data structure as a log of operations in the DHT, with a separate log per client. That is, an application using L^* does not directly store its data structure in the DHT; instead, the data structure is implied by the history of operations in the logs, and L^* stores log records in the DHT. Clients communicate through L^* and the DHT; they do not directly talk to each other or any single server. A client updates the data structure by appending records to its log; a client reads the current state of the data structure by scanning all clients' logs. Logging allows clients to perform complex operations atomically with respect to client failure. Logging operations, use of a log for each client, and deterministic log ordering mean that concur-

rent updates to the same data produce some acceptable outcome reflecting the operations, rather than a corrupted data structure.

The heart of L^* is its algorithm for resolving the order of log records in different clients' logs. This algorithm deterministically produces a single ordering of log records. That is, L^* always chooses the same order for every two log records for all clients. This property means clients agree on the order of completed updates, even if those updates were issued concurrently.

At a higher level, applications use the L^* API to implement consistent data structures. When the data structure is quiescent, L^* guarantees that clients agree on the state of the data structure. L^* optionally provides mutual exclusion for applications that need to ensure atomicity for multi-step operations. Applications benefit from being able to choose which consistency model to use; strong consistency incurs higher cost and is typically not necessary.

We built a multi-user peer-to-peer read-write file system, Ivy [6], that uses L^* to store all file system data and meta-data. The use of per-participant logs allows Ivy to support concurrent updates to the file system without using locks, and yet still maintain meta-data consistency. Ivy implements most file system operations without mutual exclusion; the only exceptions are file and directory creation. File and directory creation require mutual exclusion to avoid duplicate files or directories. Despite its use of logs, L^* makes it easy to build applications with good performance; Ivy caches aggressively, and checks the validity of the whole cache just by checking whether any logs have changed recently.

Section 2 describes DHash, the DHT on which L^* is layered. Section 3 describes the structure of per-participant logs and L^*'s API. Section 4 describes how L^* maintains consistent data structures. Section 5 describes how L^* deals with stale-data attacks from malicious DHash servers and network partition. Section 6 presents an example use of L^* to construct a serverless, multi-user, read/write file system. Section 7 discusses related work and Section 8 concludes.

2 DHash

L^* stores all its logs in DHash [1]. DHash is a distributed peer-to-peer hash table mapping keys to arbitrary values. DHash stores each key/value pair on a set of Internet hosts determined by hashing the key. This paper refers to a DHash key/value pair as a DHash block. DHash replicates blocks to avoid losing them if nodes crash.

DHash ensures the integrity of each block with one of two methods. A *content-hash block* requires the block's key to be the SHA-1 cryptographic hash of the block's value; this allows anyone fetching the block to verify the value by ensuring that its SHA-1 hash matches the key. A *public-key block* requires the block's key to be a public key, and the value to be signed using the corresponding private key. DHash refuses to store a value whose hash or signature does not match the key. L^* checks the authenticity of all data it retrieves from DHash. These checks prevent a malicious or buggy DHash node from forging data, limiting it to denying the existence of a block or producing a stale copy of a public-key block.

DHash offers a simple interface: put(key,$value$) and get(key). L^* assumes that, within any given network partition, DHash provides write-read consistency; that is, if put(k,v) completes, a subsequent get(k) will yield v. The current DHash implementation provides write-read consistency except when partitions are healing; however, potential solutions to this problem exist [2].

DHash assumes that only one writer of a public-key block is active at a time. Each public key block includes a sequence number which DHash uses to prevent overwriting newer data with stale data. Furthermore, for concurrent put(k,v) and get(k), get(k) returns either the value before or after put(k,v).

L^* is designed to also work with other untrusted network storage technologies with similar properties, such as PAST [11], Tapestry [16], or Kademlia [4].

3 Per-participant Logs

L^* represents a data structure using a set of logs, one log per participant. A log describes all of one participant's changes to the data structure. Each participant appends only to its own log, but reads from all logs.

L^* uses DHash content-hash blocks to store log records. Each log record contains the DHash key of the previous log record in the participant's log. A log record is immutable; if a log record were changed, its content-hash, and hence its DHash key, would have to change as well. L^* stores the DHash key of a participant's most recent log record in a mutable DHash public-key block, called the *log-head*. Thus, a participant's log can always be obtained from the key used to store the participant's log-head. Each user of a data structure may have multiple key pairs and log-head blocks, one for each host that the user uses. Formally, we define a participant as follows.

Definition 1. *A* **participant** *is an entity with a public-private key pair and a log-head block. At most one instance of a given participant can be active at a time.*

Table 1 describes fields that appear in log-heads and log records. The $prev$ field contains the previous record's DHash key. The seq field is an incrementing per-log sequence number. The $version$ field is a version vector [8] that L^* uses to decide how to interleave mul-

Field	Use
$prev$	DHash key of next oldest log record
seq	per-log sequence number
$version$	version vector
$head$	DHash key of the log-head

Table 1: Fields in all L^* log-head objects and log records.

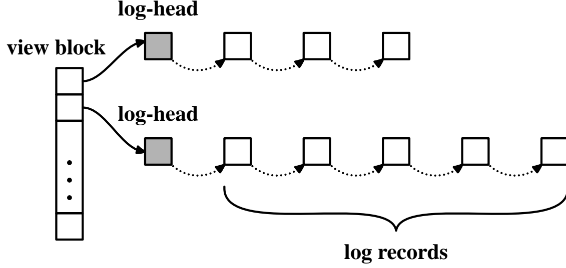

Figure 1: Example of a L^* view and logs. White boxes are DHash content-hash blocks; gray boxes are public-key blocks.

tiple logs. The $head$ field contains the DHash key of the log-head.

Participants that share a data structure agree on a *view*: the set of logs that comprise the data structure maintained by that application. A view is stored in a *view block*, a DHash content-hash block containing pointers to all log-heads in the view. A view block with a given key is immutable; when a data structure's participants decide to accept a new participant, they must all make a conscious decision to trust the new participant and to adopt a new view block, with a new key, that includes the new participant's log. The lack of support for automatically adding new participant to a view is intentional.

L^* uses the view block key to verify the view block's contents. The contents are the public keys that name and verify the participants' log-heads. A log-head contains a content-hash key that names and verifies the most recent log record. It is this reasoning that allows L^* to verify it has retrieved correct log records from the untrusted DHash storage system. Figure 1 summarizes the structure of per-participant logs and view block.

L^* provides an API that applications use to access logs. A participant modifies the data structure by appending new log records to its log, then changing the log-head to point to the newest log record. Multiple participants can modify the data structure concurrently without acquiring locks; each participant only modifies its own log-head. A participant constructs a response to a query on the data structure by reading all the logs. To avoid the expense of repeatedly reading the whole log, participants can create snapshots summarizing the data structure.

L^* needs to impose an order on log records from different logs. The order should obey causality (i.e. if an update A completes before another update B, A is ordered earlier than B) and should be the same for all participants, even for concurrently created log records. L^* creates such an order using the version vector in each log record.

3.1 Combining Logs

Each log record includes two pieces of information that are later used to order the record. The seq field contains a numerically increasing sequence number; each log separately numbers its records from zero. The $version$ field is a version vector. A log record r's version vector records pointers to the most recent record in each log at the time that r was created.

Each vector contains a tuple *(u,v)* for each log in the view (including the participant's own log). *u* is the DHash key of the log-head of the log being described, and *v* is the DHash key of that log's most recent record at the time the version vector is created. L^* saves DHash keys rather than just sequence numbers so it can recover from corrupted logs and from a malicious participant retroactively changing its log by pointing its log-head at a newly-constructed log. For simplicity, the rest of this paper replaces *u* with the name of the participant and *v* with a numeric value that refers to the sequence number contained in the record pointed to by a tuple.

Definition 2. *For a version vector x and participant i, $x[i]$ is either the sequence number recorded in x for participant i's log, or 0 if i does not appear in x.*

Definition 3. Version vector comparison: *If x and y are two version vectors, then $x >_v y$ iff for every participant i, $x[i] \geq y[i]$, and there exists a participant j such that $x[j] > y[j]$. x and y are concurrent, or $x \approx_v y$, if $x \not>_v y$ and $y \not>_v x$. $x \geq_v y$ iff $x >_v y$, or x is y, or $x \approx_v y$.*

For simplicity, for two log records r and s, this paper uses $r >_v s$, $r \geq_v s$, and $r \approx_v s$ to expression relationship between their version vectors. For example, $r >_v s$ is short for $r.version >_v s.version$.

Because a log record contains only a pointer to the next oldest log record, L^* traverses each log in reverse chronological order, starting from the most recent log record. An applications uses L^* to read the logs record by record until it finds the information it needs.

L^* orders log records based on causality. If two log records r and s have version vectors $r >_v s$, then s must have been in a participant's log when r was created. Thus $>_v$ reflects the causality between these two

```
order (list of log-heads H, callback cb)
    list of log records R
    sort H in decreasing order by DHash key
    for (i := 0; i < H.size (); i++)
        R[i] := DHash :: get (H[i].prev)
    for (;;)
        int latest
        log record r := nil
8       for (i := 0; i < R.size (); i++)
9           if (R[i] = nil)
10              continue
11          if (r = nil OR R[i] >_v r)
12              r := R[i]
13              latest := i
        if (r = nil)
            break
        else
            int retv := cb (r)
            if (retv ≠ 0)
                return retv
            if (r.prev = nil)
                R[i] := nil
            else
                R[i] := DHash :: get (r.prev)
                if (R[i] = nil)
                    fatal ("cannot load block")
    return 0
```

Figure 2: order() interleaves multiple logs in reverse order, starting with the most recent log record. order() calls application callbacks for each log record.

log records. When participants update their logs concurrently, the new log records contain concurrent version vectors. An application must tolerate whatever order L^* chooses to impose on concurrent log records, but the application may depend on L^* always ordering any two records in the same way for all the participants. Figure 2 describes the order() procedure that, given a list of log-heads, interleaves multiple logs in reverse order, starting with the most recent log record. order() takes in a callback function from the application; order() calls this function for every log record. order() is similar to merging sorted lists.

order() works in three phases. In the first phase, order() sorts the log-heads by the DHash key of each log-head, highest key first. It then fetches the most recent log record from each log into an array R, in the same order as the log-heads. In the second phase, order() iterates through R, looking for the most recent log record r. Because R is ordered by the DHash keys of the log-heads, L^* essentially orders log records with concurrent version vectors based on their log-head keys. In the third phase, order() passes r to the callback function. If the callback function does not stop log traversal, order() fetches $r.prev$ from DHash. order() repeats the second phase until all the log records have been processed.

```
version_vector latest   // local to each participant
traverse (callback cb)
    version_vector v
    list of log-heads H
    for each participant i ∈ the current view
        h_i := DHash :: get (i.key)
        v[i] := h_i.seq − 1
        H.push_back (h_i)
    if (v >_v latest)
        latest := v
    return order (H, cb)

append (log-head h_a, list of log records R)
    for each r ∈ R
        r.seq := h_a.seq
        r.version := latest
        r.prev := h_a.prev
        r.head := h_a.head
        h_a.seq := h_a.seq + 1
        h_a.prev := SHA(r)
        latest[a] := h_a.seq − 1
        DHash :: put (h_a.prev, r)
    DHash :: put (SHA(h_a.key), h_a)
```

Figure 3: L^* API: applications use traverse() and append() to maintain their data structures.

3.2 L^* API

L^* offers a simple API with two procedures, traverse() and append(). An application uses traverse() to perform lookup operations on its data structure. It constructs a response to each lookup after traversing logs. Applications use append() to append new log records and then update the log-head. A call to append(), in essence, modifies the data structure. Figure 3 describes the traverse() and append() procedures.

A program typically modifies a data structure after performing a lookup. For each new log record, append() uses a version vector, $latest$, created by the previous traverse() call. $latest$, maintained internally by L^*, captures the most recent state of each participant's log.

Because log-head fetch requests arrive at different DHash servers at different times, when several participants concurrently update their logs, it is possible that a participant's call to traverse() initially includes only a subset of the concurrent updates. A short time later, another call to traverse() includes the remaining updates, but some of which are ordered before the first subset.

Section 4 describes how to cope with this brief period of inconsistency.

3.3 Network Partition

In the case of a network partition, L^*'s design maximizes availability at the expense of consistency by allowing updates to proceed in all partitions. This approach is similar to that of Ficus [7].

L^* is not directly aware of partitions, nor does it directly ensure that every partition has a complete copy of all the logs. Instead, L^* depends on DHash to replicate data enough times, and in enough distinct locations, that each partition is likely to have a complete set of data. Whether this succeeds in practice depends on the sizes of the partitions, the degree of DHash replication, and the total number of DHash blocks involved in an application's data structure. The particular case of a user intentionally disconnecting a laptop from the network could be handled by instructing the laptop's DHash server to keep replicas of all the log-heads and log records; there is currently no way to ask DHash to do this. When a partition does not contain all the blocks needed by L^*, L^* stops working.

When network partitions, DHash does not provide write-read consistency. A `get()` in one partition does not return the value written by a `put()` in another partition.

After a partition heals, the fact that each log-head was updated from just one host prevents conflicts within individual logs; it is sufficient for the healed system to use the newest version of each log-head. Section 5 describes recovery from partition in more detail.

4 Consistency

This section describes how L^* maintains consistent data structures. L^* interleaves multiple logs deterministically so that decentralized clients can agree on the order of completed updates, even if those updates were issued concurrently. When the data structure is quiescent, L^* guarantees that clients agree on the state of the data structure. L^* optionally provides mutual exclusion for applications that need to ensure atomicity for multi-step operations (e.g. checking if a file exists, then create it if it does not). Applications benefit from being able to choose which consistency model to use; strong consistency incurs higher cost and is typically not necessary.

This section assumes cooperating DHash servers and full network connectivity. Recall that under these assumptions, DHash provides write-read consistency.

4.1 Ordering of Log Records

An application that uses a single server or server cluster to maintain its data structure depends on the server or server cluster for data structure consistency. Typically, a single server executes operations serially, thus participants can always agree on the state of the data structure after each operation. A server cluster often guarantees that within a bounded time, distributed participants agree on the state of the data structure. It would be impossible to maintain data structure consistency unless L^* offers similar guarantees to its applications.

When multiple participants are in the middle of updating their logs, it is possible that some calls to `traverse()` see some of the updates, while others see a different set of updates. Consequently, L^* does not guarantee that participants see the same set of log records at any given time. L^* ensures, however, that `order()` passes log records to the callback function in the same order for every participant. Therefore, participants always agree on the order of completed updates even if the updates were issued concurrently. We prove this property below.

For simplicity, we use $x >_r y$ when `order()` passes x to the callback function before it passes y to the callback function. We use big X to refer to log record x's log. Recall that, in `order()`, $R[X]$ contains the most recent log record in X that `order()` has not passed to the callback. Also recall that R is sorted based on the keys of the log-heads.

Lemma 1. *If x and y are two log records such that $x >_v y$, then* `order()` *always orders $x >_r y$.*

Proof. Proof by contradiction. Assume that `order()` orders $y >_r x$. Thus at some point prior to `cb`(x), y is in R. We consider two cases, when $x.head > y.head$ and vice versa. For each case, we look at how the inner loop compares each of $R[i]$ against r (lines 8-13).

First, assume that $x.head > y.head$. When the inner loop variable i refers to y's log, the loop has already examined x's log, so $r \geq_v R[X]$. Because `cb`(x) has not been called, $r \geq_v x$. Because $x >_v y$, it is also the case that $r \geq_v y$. Hence $r \neq y$ at the end of the inner loop. Therefore $y >_r x$ is impossible. Contradiction.

Next, assume that $y.head > x.head$. For $y >_r x$, it must be that, at some point, $r = y$ when the inner loop variable i refers to x's log. Because $R[X] >_v y$ as long as `cb`(x) has not been called, $R[X]$ replaces y as the value of r, as long as `cb`(x) has not been called. Hence y cannot be ordered before x. Contradiction. □

Lemma 2. *Let x and y be two log records with concurrent version vectors. If* `order()` *orders $x >_r y$, and $y.head > x.head$, then there exists another log record z, such that $x >_v z$ and $z.head > y.head$, and $z \geq_v y$.*

Proof. Because $x >_r y$, at some point prior to $\texttt{cb}(y)$, x is in R. Because $y.head > x.head$, when the inner loop variable i refers to x's log, $r \geq_v R[Y]$. We look at three possible values of r at this point in time.

First, r is from Y. Because $\texttt{cb}(y)$ has not been called, it must be that $r >_v y$ or r is y. In this case, $r \geq_v x$, and hence $r \neq x$ at the end of the inner loop. Thus, x cannot be ordered ahead of y. Contradiction.

If r is not from y's log, either $r >_v y$, or $r \approx_v y$ and $r.head > y.head$. In the former case, because $x \approx_v y$, $r \geq_v x$, and hence $r \neq x$ at the end of the inner loop. Thus x cannot be ordered ahead of y. Contradiction.

Finally, we are left with $r.head > y.head$ and $r \approx_v y$. For $x >_r y$ to happen at some point, $x >_v r$ in one of the instances of the inner loop before we return to the first case. Thus r fits the criteria for z. □

Theorem 1. *If* `order()` *ever orders two log records x and y as $x >_r y$, then it cannot order $y >_r x$ for any participant at any time.*

Proof. From Lemma 1, if $x >_v y$ or $y >_v x$, then the theorem holds. This proof shows that when $x \approx_v y$, the theorem also holds. Without loss of generality, assume $y.head > x.head$. We will show, using proof by contradiction, that it is impossible to have both $x >_r y$ and $y >_r x$.

From Lemma 2, if $x >_r y$, there exists another log record z, such that $x >_v z$, $z \geq_v y$, and $z.head > y.head$. Because $x >_v z$, if a participant sees x, it must also see z. Otherwise we have loss of data and the system halts. [1] We examine what happens when `order()` orders $y >_r x$. Because $y.head > x.head$, at some point in time, $r = y$ when the inner loop variable i refers to x's log. Then, for all w in R such that $w.head > y.head$, $y >_v w$. But this contradicts with the existence of z, since $z.head > y.head$ and $z \geq_v y$. □

Theorem 1 implies that participants agree on the order of completed updates, even if these updates were issued concurrently. Theorem 1 also implies that after partition heals, updates issued in separate partitions are ordered deterministically as well.

4.2 Relaxed Fetch-Modify Consistency

A common consistency model that distributed systems use is fetch-modify consistency [5], which totally orders all fetches and modifies on the same object and guarantees that a fetch sees the results of all modify operations ordered before it. `traverse()` and `append()` offer similar, but slightly weaker, semantics.

[1] Because log-head writes are not atomic, before the log-head write that makes z visible completes, it is possible that a participant sees x but not z. Because x refers to z in z's log, the participant knows that a stale version of z's log has been fetched and re-tries until it sees z.

Definition 4. *The* **issue time** *of* `traverse()` *is when the participant issues the first log-head fetch request. The* **completion time** *of* `append()` *is when the log-head write completes in* `append()`*.*

Definition 5. *A call to* `append()` **occurs before** *a call to* `traverse()` *iff* `append()`*'s completion time is earlier than the* `traverse()`*'s issue time.*

Lemma 3. *If a call to* `append()` *occurs before a call to* `traverse()`*, then when* `traverse()` *calls* `order()`*,* `order()` *sees all the log records written by the* `append()`*.*

Proof. Let x be the participant that issued the `append()`. Because `append()` occurs before `traverse()`, when `traverse()` issues a fetch request for x's log-head, x's log-head has already been changed to point to the new log records. Because DHash offers write-read consistency, `order()` sees all the log records written by the `append()`. □

Lemma 3 deviates from fetch-modify consistency [5] because a call to `traverse()` may also return log records appended after the issue time of `traverse()`. Even worse, because log-head fetch requests arrive at different DHash servers at different times, when multiple participants are in the middle of updating their logs, calls to `traverse()` by different participants may return different log records. Many shared memory models offer similarly weak concurrency semantics: concurrent processes only agree on the order of updates by one process, but not on the order of updates by concurrent processes. L^* differs from these models in that while concurrent updates are *first seen* at different times, participants agree on the ordering of the updates, and therefore the final state of the data structure, eventually.

Theorem 2. *If an application uses* `traverse()` *and* `append()` *to perform operations on a data structure, then, with full network connectivity, after all updates have been completed, every participant sees an identical, up-to-date, state of the data structure.*

Proof. From Lemma 3 and Theorem 1. □

In practice, different participants typically update different parts of the data structure. If at the application level these updates do not conflict with a concurrent lookup (e.g., the update modifies files in a different directory), then Theorem 2 holds for the lookup.

Theorem 2 is adequate when operations that affect each other are issued serially. Applications that need atomicity for multi-step operations must use L^*'s mutual exclusion algorithm.

4.3 Mutual Exclusion

traverse() and append() do not provide strong concurrency guarantees. For example, a call to traverse() may not see log records written by a call to append() if append() does not occur before traverse(). As a result, concurrent updates to the data structure can take place without one noticing the effects of the others. This behavior can result in non-sequential execution traces.

Applications can cope with this weak concurrency semantics with mutual exclusion, also implemented using traverse() and append(). The mutual exclusion algorithm uses three non-data structure specific log records. A participant appends a Prepare log record to announce its intention for mutual exclusion. The Prepare specifies a *handle* that identifies a part of the data structure. A participant appends an Exclusive log record if it achieves mutual exclusion. Finally, a Cancel log record cancels the previous Prepare or Exclusive log record.

Definition 6. *A* Prepare *or* Exclusive *log record r in participant a's log is* **invalid** *iff*

1. *There is a* Cancel *log record c also in a's log, $c >_v r$, and c and r identify the same handle. Or,*
2. *N seconds have passed since r was first seen.*

Otherwise, r is **valid**.

The mutual exclusion algorithm works in two phases. In the first phase, a participant x checks if another participant wants to or already has mutual exclusion. If not, x announces its intention for mutual exclusion by appending a Prepare log record. Otherwise, x backs off for a random amount of time and re-tries. In the second phase, x checks other participants' logs again. If another participant wants to or already has mutual exclusion, then x backs off and re-tries. Otherwise, x achieves mutual exclusion and appends an Exclusive log record. The mutual exclusion algorithm assumes synchrony. That is, it does not work if network delay (i.e. latency to DHash servers) or processing delay (i.e. latency of code protected by the mutual exclusion) exceeds N seconds. This section assumes this is not the case. Figure 4 presents the pseudocode of the algorithm.

The rest of the section describes properties of acquire() and release(). For now, we assume participants only update one part of the data structure. That is, Prepare, Exclusive, and Cancel use the same handle.

Lemma 4. *If r and r' are log records of two different participants such that $r >_v r'$, then prior to* append(r')*, no* traverse() *call by the same participant calls the callback with r.*

```
acquire (handle h)
    log record p := null
    check_conflict (log record r) {
        if (r is a valid Prepare(h) or
          Exclusive(h)) and r ≠ p
            return 1
        return 0
    }
    int r := traverse (check_conflict)
    if (r = 1)
        backoff for r seconds, r := (0, 10]
        return acquire (h)
    p := Prepare(h)
    append (p)
    r := traverse (check_conflict)
    if (r = 1)
        append (Cancel(h))
        backoff for r seconds, r := (0, 10]
        return acquire (h)
    append (Exclusive(h))
    return OK

release (handle h)
    append (Cancel(h))
```

Figure 4: Participants use acquire() and release() to implement mutual exclusion. acquire() passes a callback to traverse() that checks for contention.

Proof. Let x and y be participants who wrote r and r'. Assume that prior to append(r'), there is a traverse() call by y that passed r to the callback. Hence after traverse(), $y.latest[x] \geq r.seq > r.version[x]$. If this is true, then $r'.version[x] > r.version[x]$, which contradicts with $r >_v r'$. □

Lemma 5. *Let x and y be two participants. Let e_x and e_y be x and y's* Exclusive *records. If c_x is a log record that invalidates e_x, and c_y is a log record that invalidates e_y, then one and only one of the following is true,*

1. *$c_x >_v e_x \geq_v c_y >_v e_y$. Or,*
2. *$c_y >_v e_y \geq_v c_x >_v e_x$.*

Proof. It is clear that $c_x >_v e_x$ and $c_y >_v e_y$. We show, using proof by contradiction, that $c_y >_v e_x \geq_v e_y$ is impossible. Then, by similar argument, $c_x >_v e_y \geq_v e_x$ is impossible as well.

Assume $c_y >_v e_x \geq_v e_y$ is possible. Let p_x and p_y be the Prepare records for e_x and e_y, respectively. We look at what happens in x's call to acquire().

From Lemma 4, we know that, prior to append(e_x), neither traverse() call passed c_y to the callback. This in turn implies that neither traverse() call passed e_y or

p_y to the callback, because otherwise append(e_x) would not execute.

If the traverse() call prior to append(e_x) did not pass p_y to the callback, then the completion time of append(p_y) must occur after the issue time of that traverse(). This also means that the completion time of append(p_y) must occur after the completion time of append(p_x). If this is the case, however, DHash write-read consistency guarantees that the traverse() call after append(p_y) passes p_x to the callback. Hence append(e_y) would not execute. Contradiction. □

Definition 7. *A* **critical region** *is a sequence of operations surrounded by calls to* acquire() *and* release() *that protect these operations. The critical region executes after* acquire() *succeeds. The* **duration** *of the critical region extends from the issue time of the first operation in the sequence to the completion time of the last operation in the sequence.*

The following theorem proves that acquire() and release() provides mutual exclusion for critical regions.

Theorem 3. *Assuming network and processing delays do not exceed N seconds, if X and Y are two critical regions protected by the same handle, then durations of X and Y do not overlap.*

Proof. Let the first and last operations in X be x_0 and x_1, and the first and last operations in Y be y_0 and y_1. Let e_x, c_x, e_y, and c_y be Exclusive and Cancel log records that protect X and Y. Without loss of generality, assume $c_x >_v e_x \geq_v c_y >_v e_y$ (from Lemma 5). This means x_0 is issued after append(e_x), and y_1 is issued before append(c_y). Therefore, x_0 is issued after y_1. □

5 Forking

So far this paper has focused on the semantics of L^* assuming DHash provides write-read consistency. This assumption breaks under two scenarios. First, while cryptographic techniques are useful for checking integrity of data returned from untrusted DHash servers, they do not ensure freshness of the data. An untrusted server can mount a stale-data attack [5] by serving an old copy of a log-head block. Second, participants can also receive stale data if they operate in different network partitions. We call both scenarios "forking". This section describes how to detect stale-data attacks and how to recover from forking.

5.1 Detection

A DHash server mounts a stale-data attack by serving an old copy of a log-head block. To observe what happens during a stale-data attack, suppose there are three participants, x, y, and z, and the participant's log-heads h_x, h_y, and h_z each has sequence number 3. This means the most recent log record in each log has sequence number 2. Let s_x, s_y, and s_z be the DHash servers that serve h_x, h_y, and h_z, respectively. We consider the following two cases.

First, suppose s_z mounts a stale-data attack by giving h'_z to x, where $h'_z.seq = 2$, and h_z to y and z. In effect, s_z tricks x into believing that the most recent log record written by z has sequence number 1 instead of 2. While x cannot detect this attack immediately, the attack is evident if y appends a log record to y's log, and x subsequently fetches a new h_y. Because s_y is not malicious, $h_y.prev.version[z] = 2$. x then notices that $h_y.prev.version[z] \neq h'_z.prev.seq$.

In general, a stale-data attack by some but not all of the servers can be detected by checking for inconsistencies between logs. If log records in one log disagree with another log's log-head on the most recent log records in the second log, the log-head of the second log is stale. Because log-head writes are not atomic, a participant can also temporarilly fetch stale log-heads in absence of a stale-data attack.

Next, consider an attack that involves every DHash server that stores a log-head. For example, suppose s_x, s_y, and s_z collude so that s_x and s_y return h'_x and h'_y to z, where $h'_x.seq = 2$ and $h'_y.seq = 2$, and the latest copy of h_x and h_y to x and y, and that s_z returns h'_z to x and y, where $h'_z.seq = 2$, and the latest version of h_z to z. x, y, and z's logs remain consistent because the attack partitions all of x and y's updates from z, and vice-versa. Fortunately, such an attack can be detected using out-of-band communication, such as e-mail notification after updates. This scenario is similar to that described in [5].

5.2 Recovery

After stale-data attacks or network partition merge, participants see all the log records written during the fork, but most have concurrent version vectors. L^* orders such version vectors using order(), so participants will agree on the state of the data structure after the partition heals.

Assuming that a participant writes only in one partition, a data structure's meta-data, the set of per-participant logs, remains internally correct after the partition heals. That is, log records that appear in logs before the partition or added during the partition remain accessible after the partition.

At the application level, however, some partitioned updates may have affected program correctness. L^* leaves conflict detection and resolution to the application; it only notifies the application when it sees log records with concurrent version vectors.

6 Experience

We built a multi-user peer-to-peer read-write file system, Ivy [6], using L^*. Each Ivy log record contains information about a single file system modification. For example, a Link log record contain information such as "link file foo into directory bar". To avoid unnecessary conflicts from concurrent updates by different participants, Ivy log records contain the minimum possible information. For example, a Write log record describes data written to a file. Each Write record contains the newly written data, but not the file's new length or modification time. These attributes cannot be computed correctly at the time the Write record is created, since the true state of the file will only be known after all concurrent updates are known. Ivy computes that information incrementally when traversing the logs.

Ivy uses traverse() and append() to implement most file system operations. To answer a lookup, Ivy calls traverse(), stopping scanning the log once it has gathered enough data to handle the request. For example, to perform a directory listing, Ivy accumulates all file names from relevant Link log records, taking more recent log records that remove or rename files into account. Ivy modifies the file system using append(). Most modify operations follow lookups. For example, prior to creating a new file, Ivy checks if the file exists already.

Ivy implements most file system operations without mutual exclusion. This design choice does not affect program correctness when users use these operations to modify different files or directories. Concurrent updates to the same file or directory, however, may result in non-sequential execution history. For example, if one program issues rename(f1,f2) while another program concurrently issues unlink(f1), both operations may succeed. If these two operations execute sequentially, one fails. In either case, however, the file system remains consistent; it looks as if the system calls were correctly executed in one order or the other.

Ivy uses mutual exclusion to implement file and directory creation (Figure 5). File and directory creation require strong concurrency semantics so programs cannot create duplicate files or directories. Also, applications can create lock files to serialize conflicting updates, such as the concurrent rename and unlink described above.

Ivy achieves good performance [6] through aggressive client-side caching. Each participant's Ivy software caches the entire state of the file system. Use of logs allows Ivy to easily validate an entire cache; if the log-heads have not changed since the cache was updated, the cache is up-to-date. A typical Ivy operation involves fetching log-heads from DHash, fetching new log records (if any), and then completing the operation entirely from the local cache.

```
create (string n, handle dir)
    check_exists (log record r) {
        if file or directory named n exists
            return 1
        return 0
    }
    acquire (dir)
    int r := traverse (check_conflict)
    if (r = 1)
        release (dir)
        return EXISTS
    R := list of log records to create n in dir
    append (R)
    release (dir)
    return OK
```

Figure 5: Ivy uses mutual exclusion to implement file creation. Applications then create lock files to serialize operations to the same file or directory.

7 Related Work

Sprite LFS [10] represents a file system as a log of operations, along with a snapshot of i-number to i-node location mappings. LFS uses a single log managed by a single server in order to to speed up small write performance. L^* uses multiple logs to let multiple participants update a data structure without a central server or server cluster.

Existing systems, such as Bayou [14] and Conit [15], have explored the idea of merging operation logs from multiple clients in order to resolve concurrent updates to a data structure. The novel contribution of L^* is to use this idea to implement real-time access to a shared data structure.

Bayou [14] represents changes to a database as a log of updates. Each update includes an application-specific *merge procedure* to resolve conflicts. Each node maintains a local log of all the updates it knows about, both its own and those by other nodes. Nodes operate primarily in a disconnected mode, and merge logs pairwise when they talk to each other. The log and the merge procedures allow a Bayou node to re-build its database after adding updates made in the past by other nodes. As updates reach a special primary node, the primary node decides the final and permanent order of log entries. L^* differs from Bayou in a number of ways. L^*'s per-client logs allow nodes to trust each other less than they have to in Bayou. L^* uses a distributed algorithm to order the logs, which avoids Bayou's potentially unreliable primary node. L^* ensures that updates leave the

data structure consistent, while Bayou shifts much of this burden to application-supplied merge procedures. Finally, L^*'s design focuses on providing useful semantics to connected clients, while Bayou focuses on managing conflicts caused by updates from disconnected clients.

TDB [3], S4 [13], and PFS [12] use logging and (for TDB and PFS) collision-resistant hashes to allow modifications by malicious users or corrupted storage devices to be detected and (with S4) undone; L^* uses similar techniques.

8 Conclusion

This paper presents L^*, a set of techniques for maintaining consistent data structures in DHTs. L^* represents the data as a log of operations in the DHT, with a separate log per participant. Participants communicate through L^* and the DHT; they do not directly talk to each other or any single server. A participant updates the data structure by appending records to its log; a participant reads the current state of the data structure by scanning the other participants' logs. Log structure, and use of a log for each participant, means that concurrent updates to the same data result in new log records in multiple logs, rather than a corrupted data structure.

L^* interleaves multiple logs deterministically so that decentralized clients can agree on the order of completed updates, even if those updates were issued concurrently. When the data structure is quiescent, L^* guarantees that clients agree on the state of the data structure. Applications can also implement mutual exclusion using L^* to achieve stronger concurrency semantics.

We built a multi-user peer-to-peer read-write file system, Ivy, that uses L^* to store all file system data and meta-data. With aggressive client-side caching, Ivy achieves good performance.

References

[1] F. Dabek, M. Frans Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. of the ACM Symposium on Operating System Principles*, October 2001.

[2] N. Lynch, D. Malkhi, and D. Ratajczak. Atomic data access in content addressable networks. In *Proc. of the First International Workshop on Peer-to-Peer Systems*, March 2002.

[3] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, pages 135–150, October 2000.

[4] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Proc. of the First International Workshop on Peer-to-Peer Systems*, March 2002.

[5] D. Mazières and D. Shasha. Building secure file systems out of Byzantine storage. In *Proc. of the Twenty-First ACM Symposium on Principles of Distributed Computing (PODC 2002)*, July 2002.

[6] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[7] T. Page, R. Guy, G. Popek, and J. Heidemann. Architecture of the Ficus scalable replicated file system. Technical Report UCLA-CSD 910005, 1991.

[8] D. Parker, G. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. In *IEEE Transactions on Software Engineering*, volume 9(3), pages 240–247, 1983.

[9] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, pages 161–172, August 2001.

[10] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[11] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Proc. of the ACM Symposium on Operating System Principles*, October 2001.

[12] C. Stein, J. Howard, and M. Seltzer. Unifying file system protection. In *Proc. of the USENIX Technical Conference*, pages 79–90, 2001.

[13] J. Strunk, G. Goodson, M. Scheinholtz, and C. Soules. Self-securing storage: Protecting data in compromised systems. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, pages 165–179, October 2000.

[14] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. of the ACM Symposium on Operating System Principles*, pages 172–183, December 1995.

[15] H. Yu and A. Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems*, 20(3):239–282, August 2002.

[16] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, April 2001.