

CFS & Oceanstore  
Peer-to-peer File Systems.

Mikkel Fischer Christensen  
&  
Jacob Johan Jensen

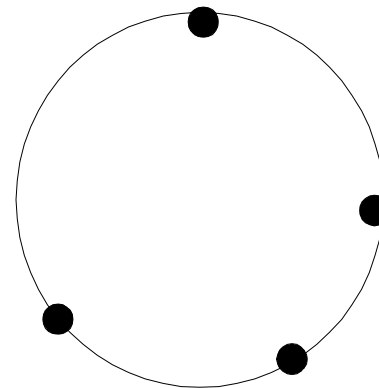
# CFS Features

- Decentralized control
- Scalability
- Availability / Persistence
- Load Balance
- Quotas
- Efficiency

# Structure

Layer	Responsibility
FS	Interprets blocks as files; presents a file system interface to applications.
DHash	Stores unstructured data blocks reliably.
Chord	Maintains routing tables used to find blocks.

ID circular space  
of 160 bit.



# Chord

We heard of Chord last week. The changes made to Chord include slightly different lookup process.

The inclusion of a succesorlist of length  $r$ .

The implementation of “locality awareness” into chord.

# Dhash

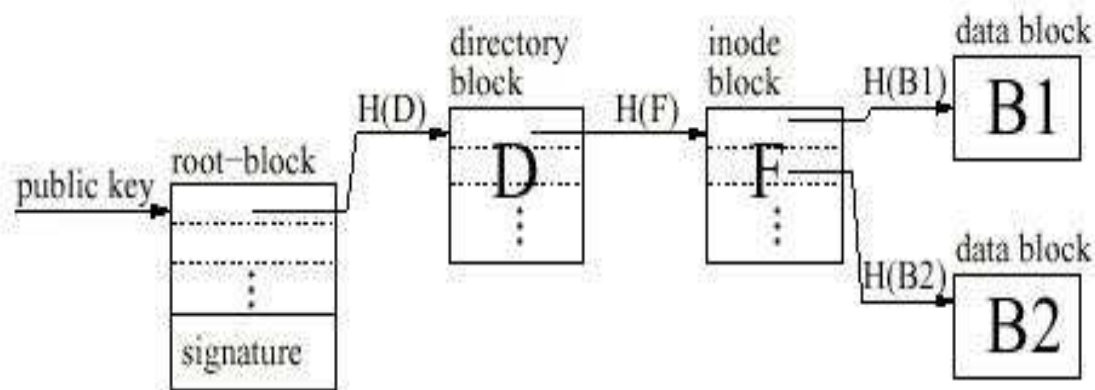
Is the block layer of the filesystem. Dhash supports 3 simple operations. Compared to XMLstore `put_s` is the main difference, because we now can overwrite a block.

Function	Description
<code>put_h(block)</code>	Computes the block's key by hashing its contents, and sends it to the key's successor server for storage.
<code>put_s(block, pubkey)</code>	Stores or updates a signed block; used for root blocks. The block must be signed with the given public key. The block's Chord key will be the hash of pubkey.
<code>get(key)</code>	Fetches and returns the block associated with the specified Chord key.

# Application

Used as a normal read only filesystem. Updates can only be made by publisher. Notice that the root block of the filesystem is the only block type you are able to overwrite.

The filesystem is build as a ordinary unix filesystem, where inode/dblock/datablock translates into dhash objects retrieved by chord.



# Application

The root block identifier ( is the SHA-1 of a publishers public key.

All other blocks are read only and use the content hash of a block as identifier in CFS.

Creation is done by publishing a directory into the CFS filesystem. Other people can mount this directory (read only) by obtaining the publishers public key.

Updates can be done by republishing, only changed blocks will have a new content hash. The root block id remains the same.

# Application

1) First choose to publish a directory into CFS:

```
halte@somehost>chordplsh -k secretkeyfile -d ./public-documents  
chordpblsh: filesystem exported as alsjsbshsddd123sha
```

2) Later mount it:

```
ln -s /sfs/chord:alsjsbshsddd123sha ~halte/public-documents-cfs
```

# Decentralization

There is no administrative tasks requiring global knowledge.

The only administrative tasks are stabilization and replication maintenance. These administrative tasks only include communication with limited amount successors and predecessors.

# Availability

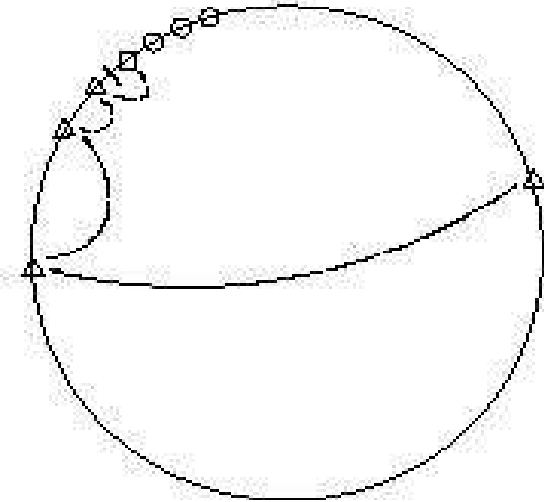
Each server maintains a list of  $r = \log(N)$  successors.

This is used in the lookup process and the replication process.

This makes it unlikely that the “circle” will be broken.

# Availability

When inserting an object into the CFS system replicas are stored at  $k \leq r$  successors of the objects id. At all times Dhash maintains  $k$  replicas as servers come and go.



\* Circles are replicas, squares the original data, and triangles is caches of the object.

# Availability

The number of replicas  $k$  means that the probability  $P$  of losing a datablock completely can be formulated as:

$$P = (1/2)^k$$

Assuming availability of a server is 50%. And  $k=16$  the probability of successful object lookup is 99,998%.

(This is based on  $N=65536$  servers, and  $k = \log(N)$ )

# Quota

It is possible to use IP based quota with the CFS system.

This is done in a decentralized approximative way. Each CFS server could allow an ip number a maximum of 0.1% of its Storage.

This form of quota does only help to avoid overusage by a single IP.

# Quota

Since there is no connection between publishers id (the public key) and the generated keys, it is currently impossible to make quota publisher based.

It is our opinion that the quota described is only useful to prevent malicious usage i.e. preventing denial of service attacks.

# Caching.

Each server allocates a portion of its storage capacity used for LRU caching of objects.

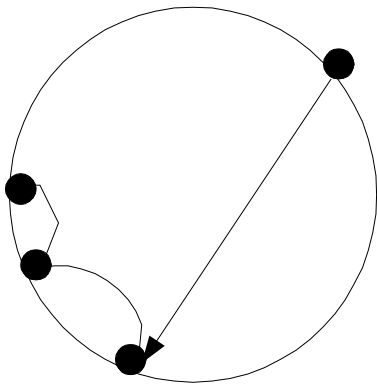
Caching is done in cooperation with the lookup process.

The last server which did not have a copy of a referred object, receives a copy of the object for its cache.

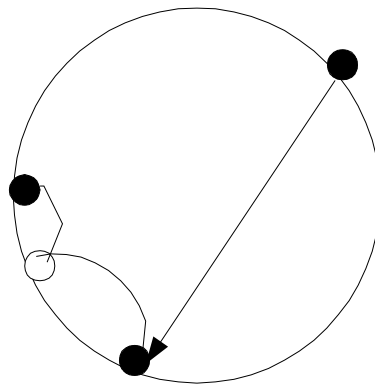
This spreads copies of referenced objects along the lookup path, closer and closer to the referee.

# Caching illustration

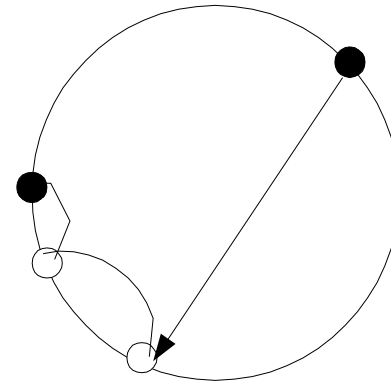
Lookup Path



Caching Path  
(first lookup)



Caching Path  
(second lookup)



# Lookup

## **The Lookup process.**

Differs from the description of lookup procedure in Chord.  
Mainly because of the inclusion of the successor list.

Is implemented iterative, but could be made recursive.

Works by choosing a node from the succesor table or the finger table as the next node in the iterative loop. The next node chosen is the node with the shortest (positive) distance in id space to the referred object.

# Lookup

Example of a lookup of key=12:

Node 0:

Successors: {1,2,3,4,5,6}

Finger: {1,2,4,8,16,32}

The possible nodes to consider as next-node:

{1,2,3,4,5,6,8}

Since we minimize we choose 8 as our next node. Then start over again choosing the next node with the successorstable and fingertable of node 8.

# Server Selection

## Server selection

Nodes in CFS are spread evenly (or random) around the world. Since Chord has no built-in preference to choose nodes with low latency. CFS has implemented an heuristic called “Server Selection” which is used in a lookup process that favors low latency nodes.

$$C(n_i) = d_i + \bar{d} \times H(n_i)$$
$$H(n_i) = \log \left( \frac{n_i - id}{2^{160}} \times N \right)$$

$H(n_i)$ : Expected number of hops from current node

$C(n_i)$ : Expected total time for cost to jump to node

# Server Selection

Example of a lookup of key=12:

Node 30:

Successors: { ...(32,10)... }

Finger: { (32,10),..., (64,30)... }

The (x,y) pairs represent node (id,latency).

The result of calculating cost  $C(n_i)$  (with hop average =20 ,

Total number of servers=12, and id space=128)

$\{(n_i, C_i)\} = \{ \dots(32,68), (64,72)\dots \}$

We now minimize over total expected cost and choose 32 as our next node. Then start over again choosing the next node with the successorstable and fingertable of node 32.

# Performance

## **Load Balancing**

Consistent hashing ensures objects are evenly stored around the id “circle”. This means no particular server is overloaded with objects.

Virtual servers can be used to balance that some clients/servers has more storage and bandwidth available. Virtual servers on the same machine can consult each others cache, finger tables, and succesor list.

Questions about CFS?

OceanStore

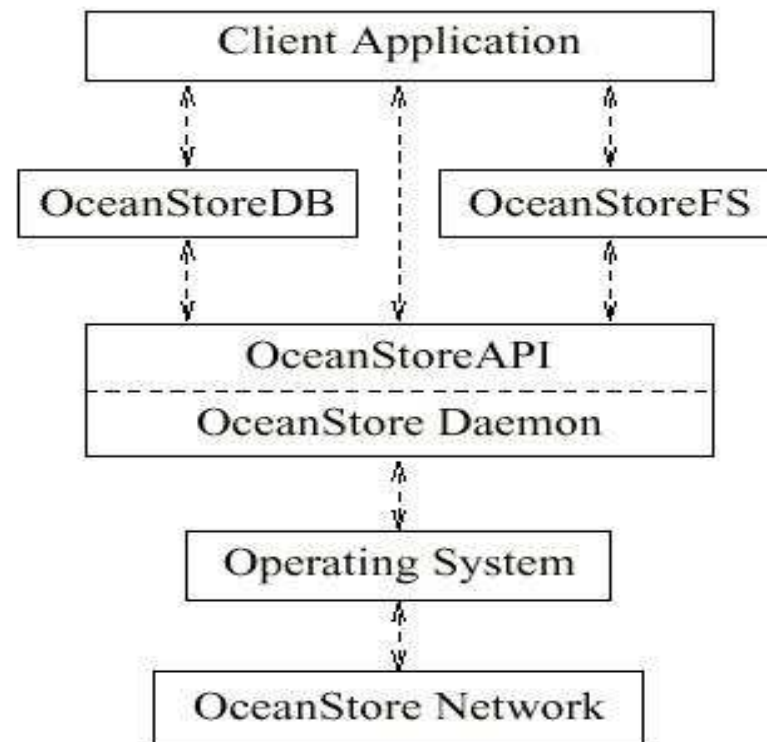
# OceanStore

A very different Peer to Peer file system with very ambitious goals.

- Support 1 Billion users.
- Each with 10000 Files.
- Lan performance.
  
- Build upon untrusted infrastructure (everything is encrypted)
- Should support nomadic data. (everything is cached)

# Application

Oceanstore filesystem offers a flexible API on which upon further API's can be build. This means that Oceanstore can be used as a UNIX file system, but also as a transaction based ACID filesystem.



# Examples of use

**Email:** OceanStore provides a storage and distribution system that can easily replace traditional e-mail services. Users can receive their e-mail within OceanStore by reserving an OceanStore object as their inbox clients.

**Database:** OceanStore's update mechanism support ACID properties.

**Multimedia:** When objects are updated, new information is rapidly disseminated via push-based mechanisms. Extremely weak update semantics may be used on streams, permitting efficient pipelining of updates.

# System overview

The fundamental unit of information in OceanStore is the persistent object. Objects are not fragmented like in CFS, objects are wholefile.

Objects is named by GUID's which is 160-bit SHA-1 hashes.

GUID's as in CFS are used to identify nodes as well as data.

Objects can be read only, the objects GUID is the secure hash of the objects content.

Objects can be writeable and the GUID is the combined hash of a human readable name and a public key.

Objects can link to other objects.

# System overview

## **Access Control.**

Objects have access control lists, more specific they contain a list of readers and writers.

Restricting readers: encrypt all data in the system and distribute the encryption key to those users with read permission.

Restricting writers: writes are signed, so that well-behaved servers and clients can verify them against an access control list.

# System overview

Objects are initially stored in the network mesh as described in the presentation of Tapestry.

Replicas of a stored object is also stored at  $k$  other servers, based on a global known 'salt' value of the objects GUID. This means if we cannot reach the root node we can obtain a replica from another server.

# Replicas

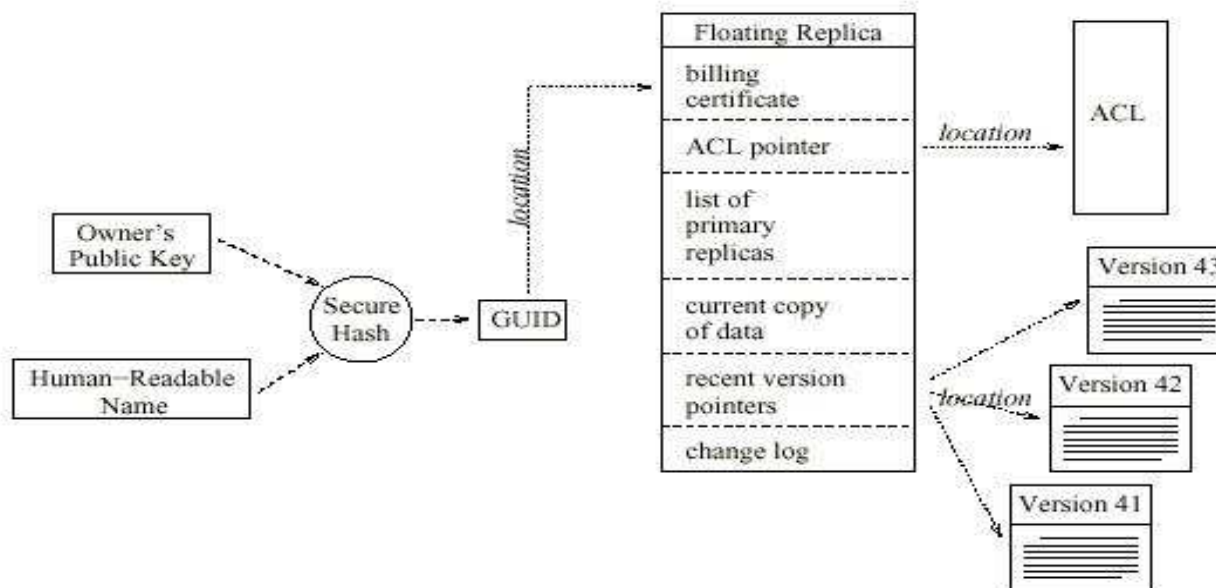
Replicas is a key component of Oceanstore. An replica is a identical copy of an object.

Primary tier replicas are the original object and the  $k$  “salted” replicas.

Secondary tier replicas are also called floating replicas, and is any object cached at any server.

# Floating Replicas

- Objects which are being read or written must be in an active state, i.e. composed of geographically distributed floating replicas.
- Floating replica contains a complete copy of the object's data, its meta-data, and logs of uncommitted updates.
- Replica is not tied to a particular physical server.
- Floating replicas are organized into primary and secondary tiers on a per-object basis and cooperate with one another to provide consistent update



# Caching

The motto of Oceanstore is cache anything anywhere.

At publishing of an object, lookup references are stored in the lookup path. This means that the next lookup will know where data resides faster, assuming the lookup path is “crossed”.

Objects are stored in caches on retrieval, but also **placed** on servers which have been source of queries for an particular object.

# Routing

Oceanstore uses two methods for routing.

Probabilistic algorithm:

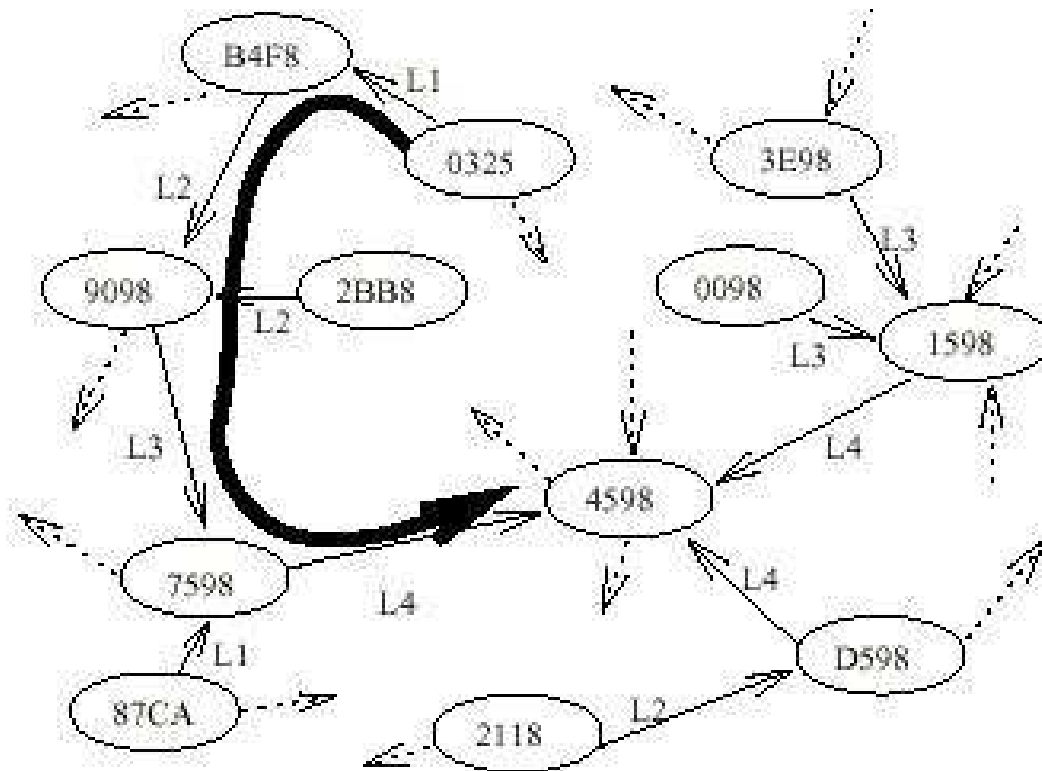
- Fast probabilistic method which uses attenuated bloom filters.
- Used to route the query to a likely neighbor.
- Used to route to query to nearby neighbours which may have a cached copy

Deterministic algorithm:

- Slower ( $\log N$ ) deterministic method – Tapestry.  
Uses a prefixed-based labeling scheme and per-node neighbor tables to maintain a good path from every node to every object.

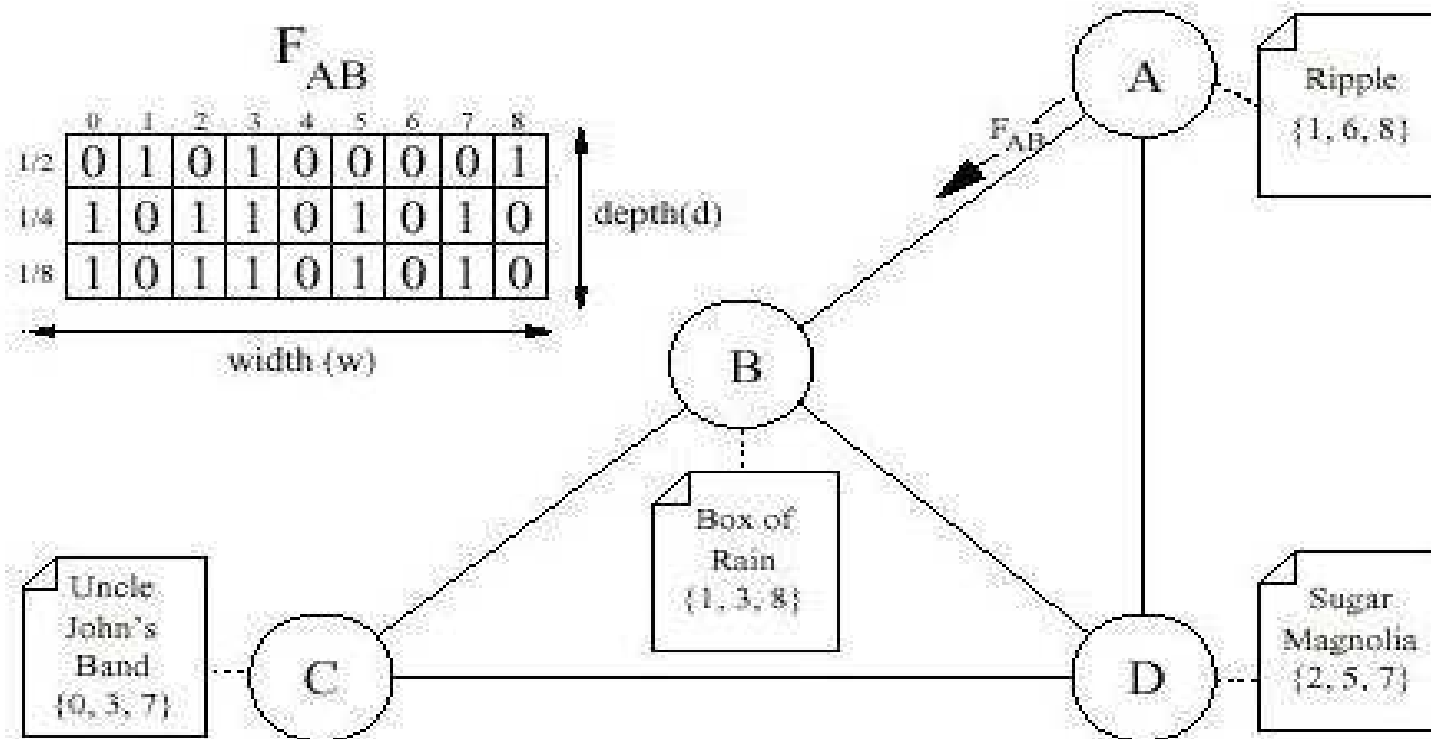
# Tapestry

The deterministic location mechanism takes a GUID and returns a pointer to the closest copy of the corresponding object.



# Bloom Filters

If a query cannot be satisfied by a server, local information is used to route the query to a likely neighbor.



# Bloom Filters

A bloom filter in the context of Oceanstore is a combined signature of all the objects on a server. This include own objects but also cached objects.

This signature is distributed to neighbours, which for every edge into the “Tapestry” mesh have a bloom filter of what objects the neighbours have.

This scheme extended to  $d$  levels of neighbours and combined with aggressive caching, we can route object queries to nearby caches.

# Update

Every update to an OceanStore object creates a new version.

The Bayou Mechanism:

- 1) To apply an update, a server first runs the dependency check against its version of the given object.
- 2) If the check succeeds, the update set is applied to the object.
- 3) Otherwise, the merge procedure is executed.

Until an update reaches a specific server, called the primary server, it is considered tentative.

The primary server serializes the update with respect to other updates on the same data object; at that point the update is said to have committed.

# Update

In an untrusted infrastructure no one server is trusted to perform commits.

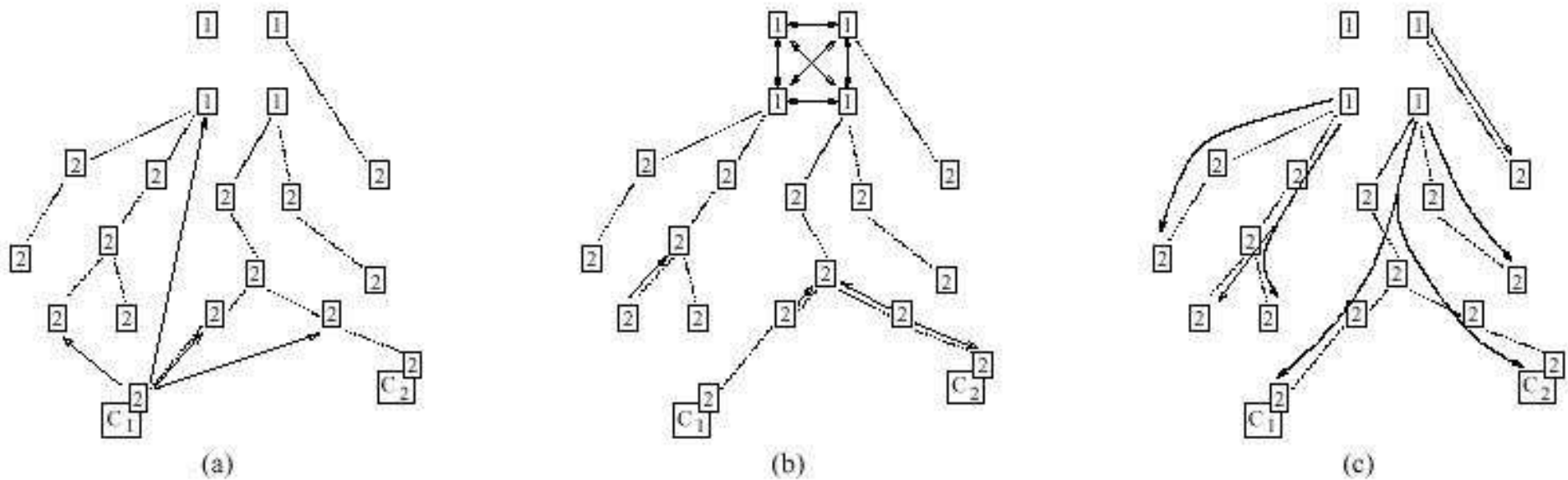


Figure 5: *The path of an update.* (a) After generating an update, a client sends it directly to the object's primary tier, as well as to several other random replicas for that object. (b) While the primary tier performs a Byzantine agreement protocol to commit the update, the secondary replicas propagate the update among themselves epidemically. (c) Once the primary tier has finished its agreement protocol, the result of the update is multicast down the dissemination tree to all of the secondary replicas.

- 1) Tier one – byzantine decision model for updates
- 2) Updating, secondary tier of replicas.

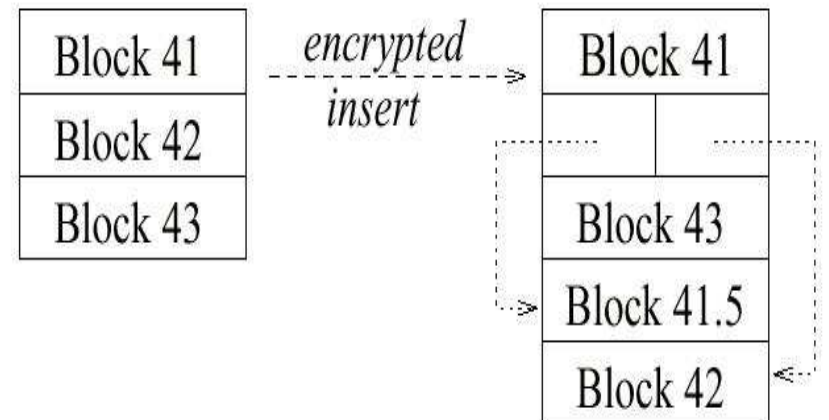
# Updating Ciphertext

Alternative predicate-update pairs:

- Compare-version(version-id)
- Compare-size(size)
- Compare-block(block-no, expected value)
- Search(search-string)

Operations applied to Ciphertext:

- Replace-block(block-no, new-value)
- Insert-block(block-no, value)
- Delete-block(block-no)
- Append(value)
- Truncate-from-start(num-blocks)



# Providing ACID semantic

The Bayou model can be used to provide ACID semantics:

- Predicate becomes the read set of a transaction
- Update becomes the write set
- Merge predicate always fails

Some applications may gain performance or availability by requiring a lesser degree of consistency than ACID semantics.

These applications are well supported by the secondary tier of replicas in OceanStore.

Secondary replicas make use of multicast or other transport mechanisms to quickly push tentative commits amongst themselves and to decide a tentative serialization order.

# Data coding model

Two distinct forms of data: active and archival

Active data: in floating replica

- Logging for updates/conflicts resolution
- Interaction with other replicas for keeping data consistent

Archival data: in Erasure-Coded Fragments

- Data coded into  $f$  fragments of which  $r_f$  is sufficient to reconstruct all data.

# Data Recovery

Active data is committed as  $n$  fragments and archival data is stored in  $2x n$  or  $4x n$  fragments introducing redundancy for easy recovery.

$$P = \sum_{i=0}^{r_f} \frac{\binom{m}{i} \binom{n-m}{f-i}}{\binom{n}{f}}$$

$P$  probability that document is available,  $n$  number of machines,  $m$  number of currently unavailable machines,  $f$  number of fragments per document,  $r_f$  number of unavailable fragments still allowing the document to be retrieved.

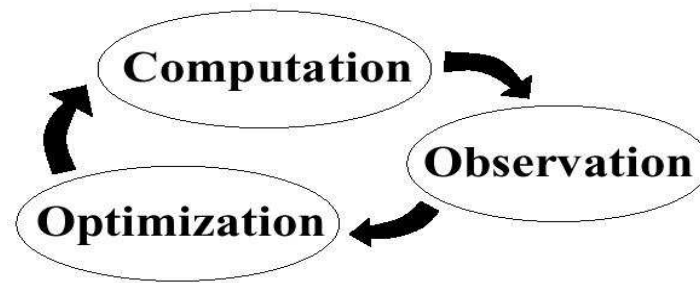
# Data Recovery – Example

A million machines, ten percent of which are currently down, simple replication without erasure codes provides only 99 % of reliability.

A 1/2-rate erasure coding of a document into 16 fragments gives the document 99.9994 % of reliability.

# Introspection

Introspection is the automated maintenance task, roughly comparable to “stabilize”. Introspection is trying to enhance durability and availability of data.



Observation modules monitor the activity of a running system and keep a historical record of system behavior.

Optimization modules use the resulting analysis to adjust or adapt the computation.

OceanStore uses introspection to select the placement and number of floating replicas, to prefetch information, and to monitor and repair the level of redundancy.

# Adaptive Replica Management

Both the number and location of floating replicas greatly affect availability and performance.

For read-only objects, this is clearly desirable. For frequently written objects, each replica increases the effort and overhead required to maintain consistency.

Floating replicas tracks client requests and measures the server load consumed when servicing these requests.

This info is used to decide whether the floating replica should migrate closer to the source of client requests or toward a group of updating clients.

# Stability and Durability Enhancement

If not properly checked, nearly any optimization in the OceanStore can cause instability. Data oscillating wildly from place to place, is one of the problems that might occur, resulting in a decreased quality of service.

To increase durability further, sweep and repair functionality is applied:

- For the global data location structure, we continually rebuild the search tree and pointers for data.
- For archival data, OceanStore servers slowly sweep through all fragments, increasing the level of replication when necessary.

Questions about  
Oceanstore?

# Our questions!

- 1) Suggest a way to implement quota's and preferably avoiding centralized control.
- 2) What effect will recursive lookup have on server selection?
- 3) Can server selection make the performance of lookup worse?
- 4) How is garbage collection implemented in CFS?
- 5) In what sense is CFS performance comparable to FTP?
- 6) What is the main difference(s) of CFS and Oceanstore?
- 7) How does Oceanstore implement query locality of objects?
- 8) Is ACID properties provided on ciphertext?