Disconnected Operations

- What is disconnected operation?
- Why support for disconnected operation?
- Approaches
- Bayou
- Coda

What is disconnected operation?

- **Disconnected operation**
  - A mode of operation that enables a client to continue accessing critical data maintained at the server during temporary interruption of network connection
- **Weakly connected operation**
  - Network condition is poor (radio signal is in lower level, connectivity is intermittent, channel noise)
  - Disconnected operation is the extreme of weakly-connected operation
- **Disconnection can be *involuntary* or *voluntary***

Why support for disconnected operation?

- **Provide data availability under mobile circumstance**
  - User mobility
  - Unpredictable network conditions
- **Client's choice:**
  - Extends battery life by avoiding wireless transmission and reception
  - Reduces network charges (high rate)
  - Allows radio silence - a vital capability in military applications.

Approaches

- **Objectives**
  - Ensure *data availability* even under disconnection
  - Tradeoff between consistency of data and autonomy of client
- **Solutions**
  - Standard file systems (e.g., NFS) are very inefficient, almost unusable
  - Approaches to achieving data availability
    - Caching, prefetching, hoarding
  - Solutions need to ensure certain degree of data consistency
    - Update conflict detection / resolution
    - Data reintegration
    - Weak consistency
      - Cost of enforcing data consistency is high

Caching

- **Places a copy of file on client to avoid requesting over network**
  - Useful for both weak connection and disconnection
- **Invalid copy may still be of limited value during disconnection (better than nothing?)**
- **Issues**
  - What to cache? (Files, objects, pages?)
  - When to cache? (Cache all newly obtained items?)
  - How to cache?
    - Simple LRU or other metrics
    - Semantic caching
    - Predictive caching

Prefetching

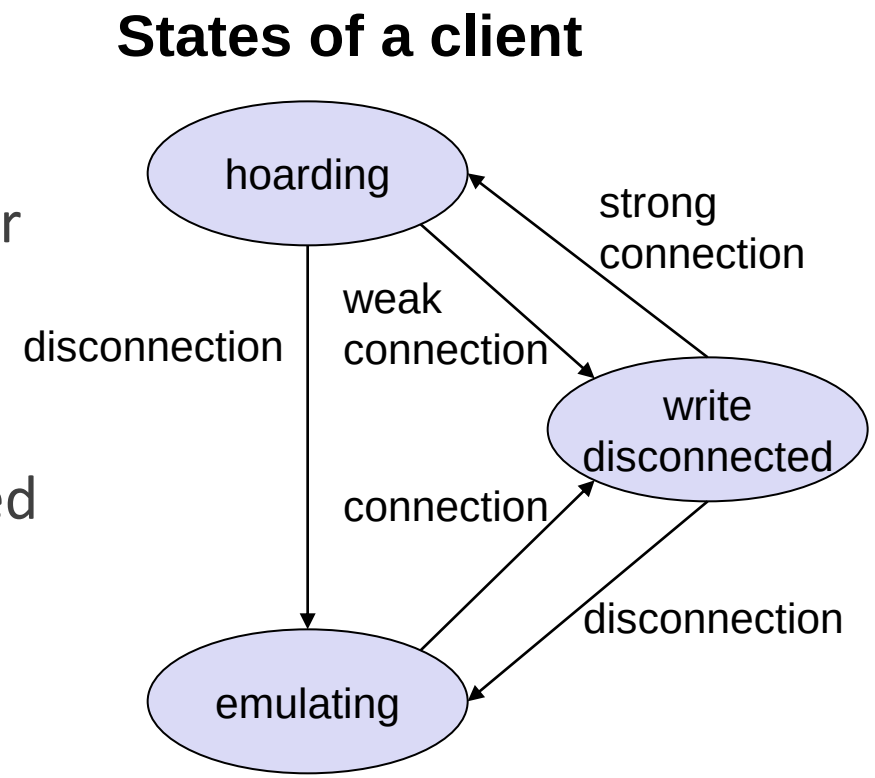
- **Anticipatory caching**
  - Cache items that may be useful in future, but not currently needed, utilizing the otherwise idle processing power or bandwidth
- **Issues**
  - What to fetch? (Files, objects, pages?)
  - When to fetch?
    - When no one uses the request channel
    - When the broadcast item is interesting
  - How to fetch?
    - Compute for the need and use the up-link channel to send request
    - Monitor broadcast channel and download the item

Hoarding

- **Pre-fetching to prepare for disconnection**
- **Issues**
  - What to hoard? (Files, objects, pages?)
  - When to hoard?
    - When planned disconnection is coming
    - With sufficient important data accumulated and high chance of disconnection
  - How to hoard?
    - User-provided information (useful for client-initiated disconnection)
    - Access pattern-based (past history)
    - Inter-file relationships

States of a client

- Three modes
  - Connected (Normal)
    - server is available, request can be satisfied from either local cache or server
  - Disconnected (Off)
    - all request must be satisfied from local cache
  - Reintegration (Recover)
    - update related info, deal with conflict etc.



Consistency

- **Main problem with strong consistency**
  - Strong consistency requires atomic updates
    - Invalidation of caches through a server
    - Difficult to achieve in mobile environments - mobile computer may not be connected to network
- **Alternative: weak consistency**
  - Tolerate occasional inconsistencies
  - Apply conflict resolution strategies subsequently
  - Use version numbering, time-stamps (content independent)
  - Use dependency graphs (content dependent)

Transparency of replication

- **Transparent replication:**
  - Allow systems that were developed to run on a central server to operate unchanged on top of a strongly-consistent replicated data storage (as seen in Oceanstore)
- **Non-transparent replication:**
  - Allow systems to be developed assuming a relaxed consistency model – applications are involved in conflict detection and resolution.
  - Hence applications need to be modified (e.g. Bayou, Coda file system etc.
    - Conflicts between application users are not easily handled transparently.
    - Applications know best on how to resolve conflicts

Non-transparent replication

- The challenge is providing the right interface to support cooperation between applications and their data managers

Bayou

- **Bayou is a system developed at PARC in the mid-90's, as the first coherent attempt to fully address the problem of disconnected operation**
- **But first, why did they call it "Bayou"?**
  - A body of water, such as a creek or small river, that is a tributary of a larger body of water.
  - A sluggish stream that meanders through lowlands, marshes, or plantation grounds.
  - Possible explanations
    - Bayous are ubiquitous, and Bayou supports ubiquitous computing
    - Bayou provides "fluid" replication
    - Allows operation when you are "bayou self"
    - Pronounced Bi-U, which makes it Ubi spelled backwards

Bayou

- Bayou is a replicated, weakly consistent storage system designed for a mobile computing environment.
- To maximize availability, users can read and write any accessible replica. (read-any/write-any)
- Bayou introduced an novel application-specific conflict detection and resolution method
- Two major design issues
  - Resolve conflicts: How?
  - Guarantee consistency: How?

The Case for Non-transparent Replication: Examples from Bayou Douglas B. Terry, Karin Petersen, Mike J. Spreitzer, and Marvin M. Theimer. IEEE Data Engineering, December 1998

Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer and Carl H. Hauser. In ACM Symposium on Operating Systems Principles (SOSP '95)

Conflict detection and resolution

- **Supporting application-specific conflict detection and resolution is a major emphasis in the Bayou design.**
- **Why application-specific?**
  - Only the application knows how to resolve conflicts, so let it specify its notion of a conflict along with its resolution policy
  - Application can do record-level conflict detection, not just file-level conflict detection
- **Split of responsibility:**
  - *Replication system*: propagates updates
  - *Application*: resolves conflict

Motivating scenario: Shared calendar

- **e.g., Sharing calendar for meeting room scheduling**
  - users can reserve room, at most one person (group) can reserve the room for any given period of time.
- **Want to allow updates offline, but then conflicts can't be prevented**
- **Let application itself speak and suggest**
  - Calendar example: resolution can be easily done at record-level

Motivating scenario: Shared calendar

- Observing updates at a coarse granularity (e.g., lock at the whole-file level) Conflict!
- Observing updates at a fine granularity (record) No Conflict
- Neither of these conclusions are warranted.

  In fact, for this application, a conflict occurs when two meetings scheduled for the same room overlap in time.

  - Reserve same room at same time: conflict
  - Reserve different rooms at same time: no conflict
  - Reserve same room at different times: no conflict

  Only the application would know this!

  So, application-specific!

Application-specific conflict detection and resolution

- **A Bayou Write not only is the typical file / database system Write (update), but also includes two mechanisms for automatic conflict detection and resolution that are intended to support arbitrary applications:**
  - Dependency checks

    information that lets each server receiving the Write decide if there is a conflict.
  - Merge Procedure (Mergeproc)

    Fix measure that server should take while encountering the conflict for this write based on app's semantics

Application-specific conflict detection and resolution

- **Dependency Checks**
  - In Bayou, application-specific conflict detection is accomplished through the use of *dependency checks*.
  - Each Write operation includes a dependency check consisting of an application-supplied query and its expected result. It works as follows:

1. run the query at a server against its current copy of the data,
2. check if the returned result matches the expected result
3. if the check fails, then the requested update is not performed and the server invokes a procedure to resolve the detected conflict

Application-specific conflict detection and resolution

- Use the meeting scheduling appplication as example

```
Bayou_Write (update, dependency_check, mergeproc) {
    IF (DB_Eval (dependency_check.query) <> dependency_check.expected_result)
        resolved_update = Interpret (mergeproc);
    ELSE
        resolved_update = update;
    DB_Apply (resolved_update);
}
```

Figure 2. Processing a Bayou Write Operation

- SQL-like query

Application-specific conflict detection and resolution

- **Merge Procedure (Mergeproc)**

  Once a conflict is detected, a *merge procedure* is run by the server in an attempt to resolve the conflict.

  - In practice, the merge procedures are written by application programmers in the form of templates that are instantiated with the appropriate details filled in for each Write.
  - The users of applications do not have to know about merge procedures

Application-specific conflict detection and resolution

```
Bayou_Write(
    update = {insert, Meetings, 12/18/95, 1:30pm, 60min, "Budget Meeting"},
    dependency_check = {
        query = "SELECT key FROM Meetings WHERE day = 12/18/95
            AND start < 2:30pm AND end > 1:30pm",
        expected_result = EMPTY},
    mergeproc = {
        alternates = {{12/18/95, 3:00pm}, {12/19/95, 9:30am}};
        newupdate = {};
        FOREACH a IN alternates {
            # check if there would be a conflict
            IF (NOT EMPTY (
                SELECT key FROM Meetings WHERE day = a.date
                 AND start < a.time + 60min AND end > a.time))
                    CONTINUE;
            # no conflict, can schedule meeting at that time
            newupdate = {insert, Meetings, a.date, a.time, 60min, "Budget Meeting"};
            BREAK;
        }
        IF (newupdate = {})   # no alternate is acceptable
            newupdate = {insert, ErrorLog, 12/18/95, 1:30pm, 60min, "Budget Meeting"};
        RETURN newupdate;}
)
```

Figure 3. A Bayou Write Operation

Example of Bayou Write mechanisms

3-tuple: <update> <dependency check> <mergeproc>

Update:

<insert, HL, 02/02/2009, 3:00pm, 1 hr, “Curriculum Meeting”>

Dependency check:

<Is there an entry in the database on 02/02/2009 at 3:00pm for 1 hr? expected answer is empty>

Mergeproc:

<Try 02/22/2009 at 4:15pm for 45min; if successful write that tuple>

- A different merge procedure could search for the next available time slot to schedule the meeting, which is an option a user might choose if any time would be satisfactory.
- Another is to allow the conflicts; sometimes users like conflicts

User-specified resolution strategies

- Classes take priority over meetings
- Faculty reservations are bumped by admin reservations
- Move meetings to bigger room, if available
- Point:
  - Conflicts are detected at very fine granularity
  - Resolution can be policy-driven

Consistency in Bayou

- **Eventual consistency**
  - All servers receive all Writes via the pair-wise anti-entropy process
  - Two servers holding the same set of writes will have the same data contents.
- **Two important features of Bayou design allow servers to achieve eventual consistency.**
  - First, Writes are performed at all servers, in the same, well-defined order.
  - Second, the conflict detection and merge procedures are deterministic so that servers resolve same conflicts in same manner.

Anti-entropy protocol

- **The protocol used among servers in Bayou for propogating Writes (updates) is called *anti-entropy* protocol**
  - Anti-entropy refers to a model of information propagation
  - A server P picks another server Q at random to exchange updates
  - Three approaches:
    - P only pushes its own updates to Q
    - P only pulls in new updates from Q
    - P and Q send updates to each other
- **This anti-entropy model guarantees that once a write reaching one of the servers, finally it will reach all of the other servers**
  - Write is propagated pair-wise between servers
  - All replicas receive all updates

Anti-entropy protocol

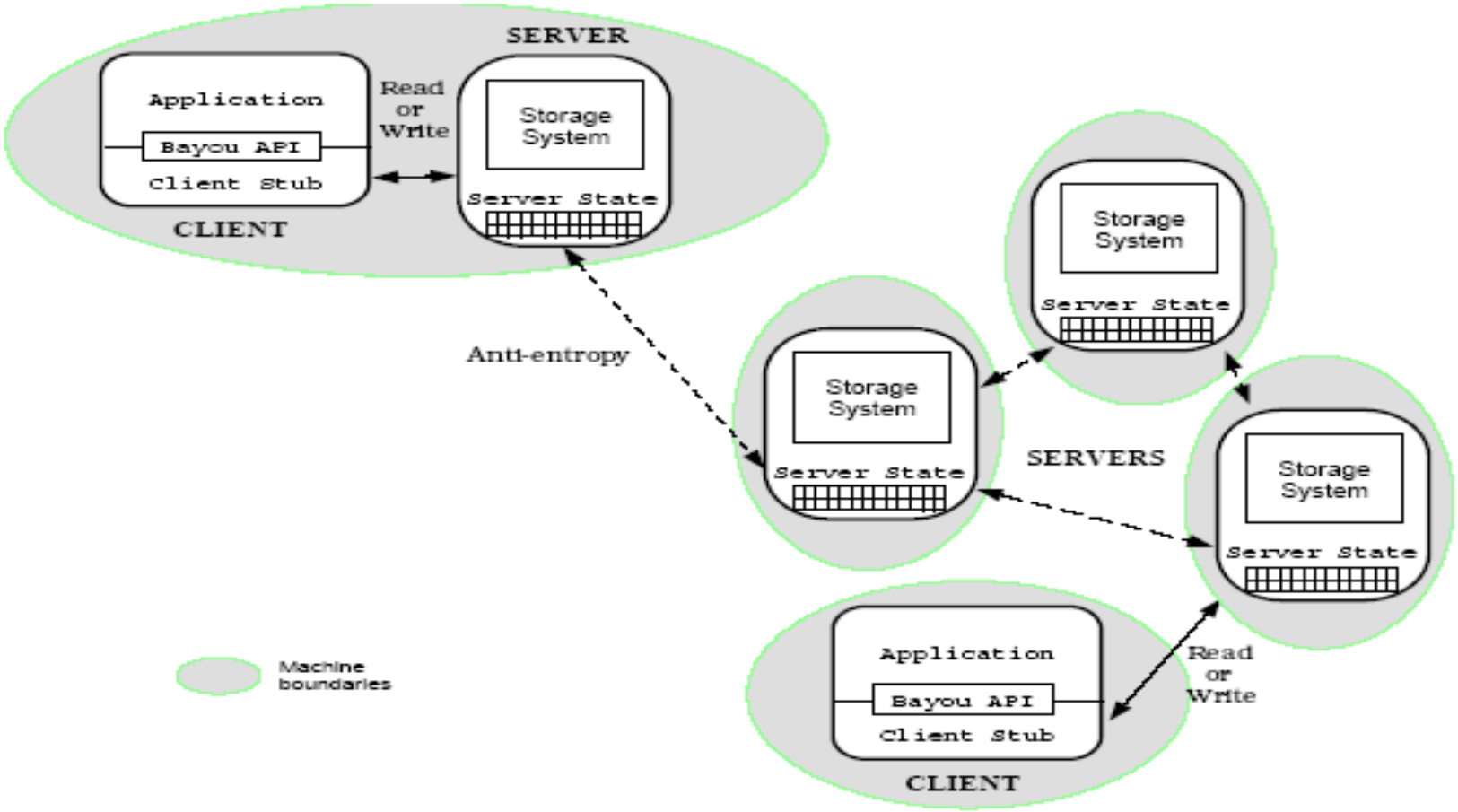


Figure 1. Bayou System Model

Ordering of updates

- **Maintain ordered list of writes at each node**
  - This will be referred to as the write log
  - Each write in the log has a unique Write ID:

    <local-time-stamp, originating-node-ID>
  - The local-time-stamp is assigned by the server that accepted the write from front end (referred to as the accepting server)

<701, A>:

Node A asks for meeting M1 to occur at 10 AM, else 11 AM in MC 316

<770, B>:

Node B asks for meeting M2 to occur at 10 AM, else 11 AM in MC 316

Ordering of updates

- Let's agree to sort by write ID (e.g., <701, A> <770, B>)
- As writes operations spread from node to node, nodes may initially apply updates in different orders
- The dependency check and merge procedures do not necessarily take care of this
- Want writes for a particular data to be performed in the same order at all replicas
  - Each newly seen write merged into write log
  - Log replayed
    - may cause calendar displayed to user to change!
    - i.e., all entries are tentative, nothing stable unless an entry is committed.
  - How do we get commits?

Criteria for committing Writes

- **For log entry X to be committed, everyone must agree on:**
  - Total order of all previous committed entries
  - Fact that X is next in total order
  - Fact that all uncommitted entries are “after” X

How Bayou agrees on total order

- **Write in Bayou has 2 states**
  - **"committed"**
  - **"tentative"**
  - When a Write is accepted by a Bayou server from a client, it is initially deemed *tentative*
  - Later the tentative will become *committed*
- **Must be able to undo writes; Why?**
  - Servers may receive Writes from clients and other servers in an order that differs from the required execution order;
  - Servers immediately apply all known Writes to their replicas.

How Bayou agrees on total order

- **Rolling back updates**
  - Keep log of updates
  - Order by some timestamp
  - When a new update comes in, place it in the correct order and reapply log of updates
  - Need to establish when you can truncate the log
  - Requires old updates to be “committed”, new ones tentative

How Bayou agrees on total order

Method to achieve well-defined order:

Divide the servers into 2 roles

- **"Primary "**
  - Be responsible for committing write
  - Set order in which data committed and propagate final committed order to secondary

    (via anti-entropy)
- **"Secondary"**
  - Accept write from user and propagate to Primary (using anti-entropy protocol)

Coda

- **A distributed file system developed at CMU that can survive network disconnection**
- **Designed on experience with using AFS:**
  - Network disconnection does occur quite often
  - Client suffer a lot when network is down
- **CODA extends AFS by**
  - Providing *constant availability* through *replication*
  - Introducing a *disconnected mode* for *portable computers*
    - Most lasting contribution
  - A good mobile file system to use

J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1):213–225, February 1992.

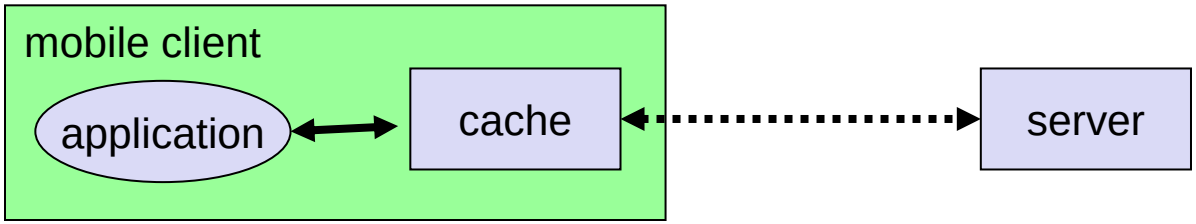
M. Satyanarayanan, J. J. Kistler, L. B. Mummert, M. R. Ebling, P. Kumar, and Q. Lu. Experience with Disconnected Operation in a Mobile Comp uting Environment. In *Proceedings of the 1993 USENIX Symposium on Mobile and Location-Independent Computing*, Cambridge, MA, August 1993.

Distributed file systems

- **File**
  - abstraction of permanent storage (e.g., disk), storing data, program, directories, etc
- **File system**
  - responsible for organization, storage, retrieval, naming, sharing, and protection of files.
- **Distributed file system: file system implemented and used on a distributed system.**
  - Files are stored at different nodes over the network and may be replicated.
  - Users can access and share the files from different sites

Client module

- **Client module hides low-level constructs of file system and directory services, making the access procedure transparent to user programs.**
  - User program should only access network files using normal file names and operations (read, write, rename, etc.)
- **Client can improve file access performance by caching frequently used files in local memory / storage.**
  - Caching should be handled by the client module and transparent to user.



Caching at client

- **The following protocols may be used:**
  - Write-through
    - An update changes local copy, also sent to server for action with similar write message
  - Delayed write
    - Update local copy and batch updates before sending them to server for action
  - Write-on-close
    - Update local copy and send final copy on closing of the file
  - Mutual-exclusive write
    - Lock the file for writing at a coordinating manager (which can be distributed)
- **In a mobile environment, additional complexities need to be considered:**
  - Weak connection: write-through and mutual-exclusive write are impractical
  - Disconnection: even delayed write may not work
- **File hoarding / prefectching and reintegration**

Andrew file system

- **AFS is a Unix-based distributed file system, runnable on Mach as well, and is compatible with NFS**
- **AFS adopts the *upload/download* model instead of merely performing remote accesses.**
  - The whole file is transferred to and cached in client sites
- **Observations based on real measurements on Unix-based systems;**
  - Most files are smaller than 10k bytes
  - Most files are of a shot live.
  - Reads are about 6 times more common than writes.
  - Sequential access is common, but random access is rate.
  - Most files are accessed by one user, and for shared files, often only a single writer.
  - Files are references in bursts. A file recently updated is likely to be accessed in neat future, in a LRU manner.

Andrew file system

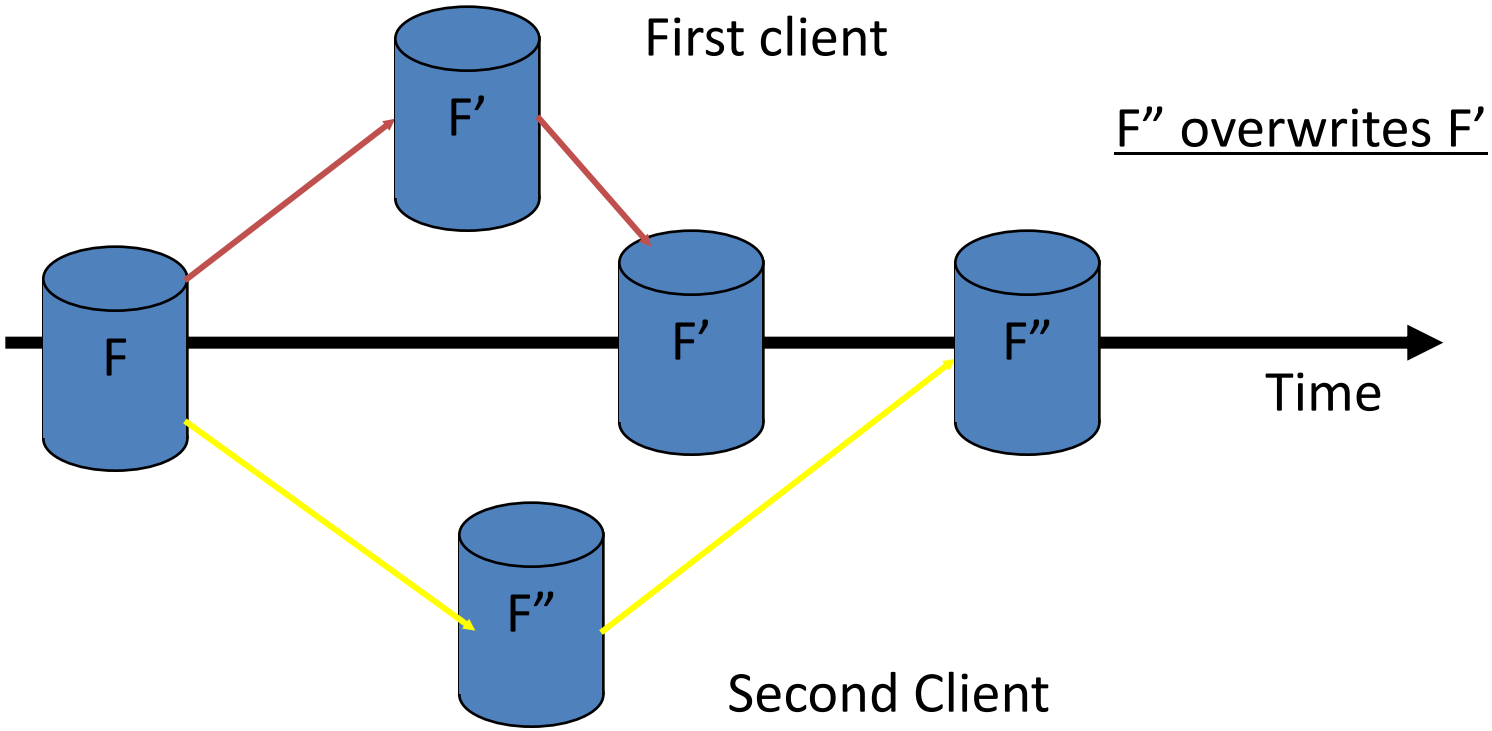
- **AFS caches a file at client upon opening the file, together with a *callback promise* token (2 states: valid or cancelled).**
  - Server *promises to notify* client whenever it receives a *new version* of the file from any other client.
    - It will send a callback to client
    - Client invalidates the cached copy, by setting the promise state to canceled.
  - Relieves the server from having to answer a call from the client every time the file is opened
    - Significant reduction of server workload

Andrew file system

- **On read/write, operation is done on local cached copy at client**
- **AFS performs write-on-close**
  - Upon a close operation, the file is sent back to server for update, retained at client if necessary
- **AFS provides *approximated* Unix semantics, close to session semantics, by using callback promise.**
  - Server is not updated until file is closed
  - Client is not updated until it reopens the file
  - Does not handle concurrent updates
  - Only final file closing operation can retain all updates it makes

Andrew file system

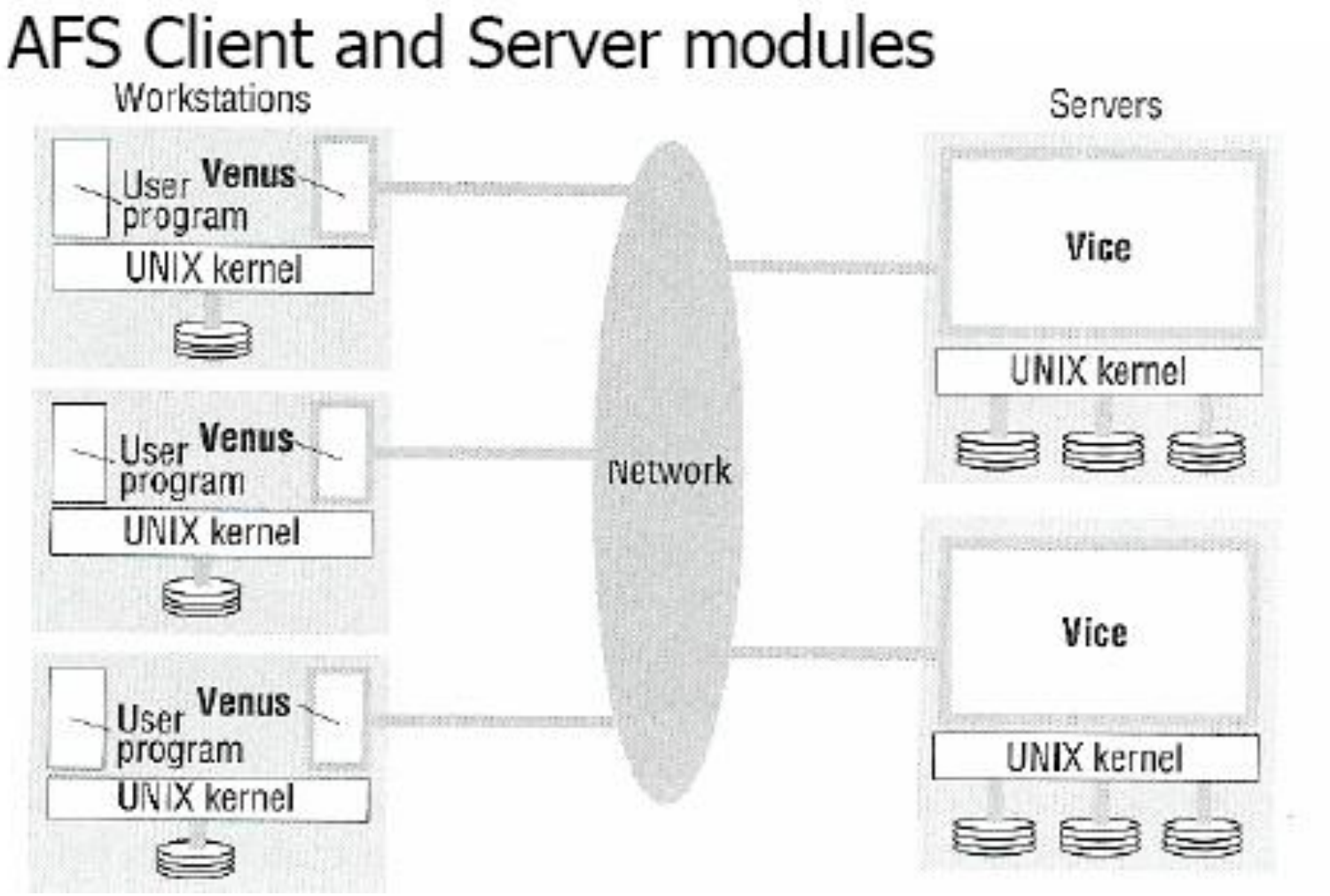
- If two users on two different workstations modify the same file at the same time, the users closing the file last will overwrite the changes made by the other user



Andrew file system

- When client crashes and recovers, it will check with server about validity of its cached files, using modification timestamp.
- To guard against missing callbacks due to message loss, callback promises must be renewed every few minutes from client.
- AFS servers are *stateful*.

Andrew file system



Andrew file system

User process	UNIX kernel	Venus	Net	Vice
open(FileName, mode)	If *FileName* refers to a file in shared file space, pass the request to Venus. Open local file and return the file descriptor to application.	Check list of files in local cache. If not present or no valid *callback promise*, send a request for the file to the Vice server that is custodian of the volume containing the file. Place the copy of the file in the local file system, enter its local name in the local cache list and return the local name to UNIX.		Transfer a copy of the file and a *callback promise* to workstation. Log the callback promise.
read(FileDescriptor, Buffer, length)	Perform a normal UNIX read operation on the local copy.			
write(FileDescriptor, Buffer, length)	Perform a normal UNIX write operation on the local copy.			
close(FileDescriptor)	Close the local copy and notify Venus that file has been closed.	If the local copy has been changed, send a copy to the Vice server that is custodian of the file.		Replace the file contents and send a *callback* to all other clients holding *callback promises* on the file.

Extension in Coda

- **In a mobile environment, weak network connection poses a major hurdle and design of propose mechanism is needed:**
  - Connection may fail intermittently, leading to high communication cost
  - Clients may become disconnected from the network
- **Coda is an extension of Andrew**
  - Its ordinary operation just stays the same as that of AFS
    - *Whole file catching*: original design purpose is reducing traffic, but also appropriate for disconnection
    - *Cache invalidation*: server's callback promise
    - *Use optimistic replication*: does not prevent inconsistency from happening but to detect it

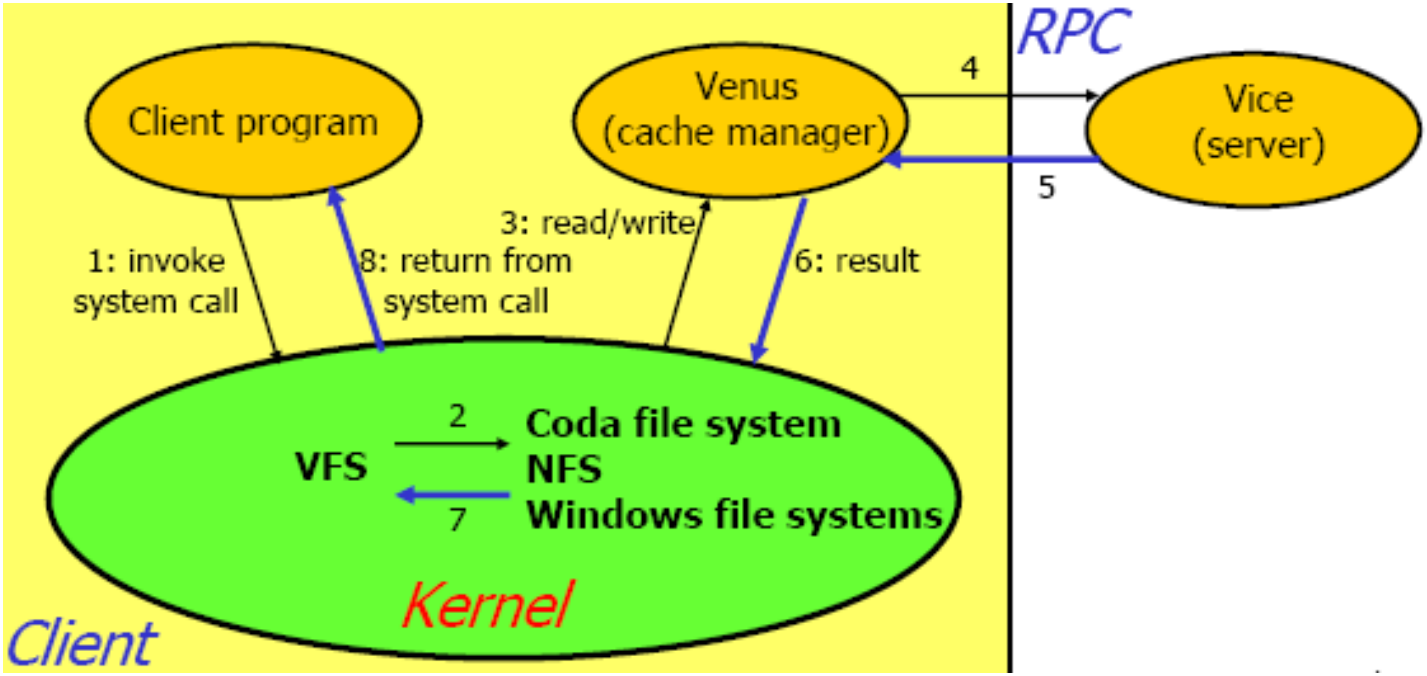
Extension in Coda

- **What makes Coda different is the disconnected mode**
- **Original design goal of Coda is to be resilient to network failures**
  - Extensive use of caching is employed
  - Supporting disconnected operation and mobility was an side effect

Coda

- **Cache whole file at client whenever possible (makes more sense to cache whole file to prepare for disconnection)**
  - Callback based caching (i.e. invalidation)
- **Automate the portable computer model**
  - Figure out which files one needs to use at home, make changes, copy back modified files next morning (auto-syn feature)
- **Use server replication to improve scalability**
  - A client reads a file from any server but writes to all servers when file is closed (read-one write-all)

Coda operation



Coda operations

- **Each client chooses a preferred server which holds all callbacks and answers all read requests from the client**
  - Can also choose the server randomly on each access
- **To serve a read, just try the local cache first**
  - If it is a cache miss, requests the preferred server for file and status.
- **To server a write, a two-phased non-blocking protocol is executed:**
  - Updated file and status are sent to all servers
  - Server receiving updates will install update and reply
  - Client send consolidated server replies in an update sent to all servers to bring them up-to-date together (this phase could be executed asynchronously or lazily to improve efficiency)

Cache consistency

- **Callbacks are used to maintain consistency**
  - When a file is accessed, a local copy is made at client and a callback is registered with server
  - If someone else modifies the file, client will get a call back from server to invalidate it.
- **Cache invalidation is used instead of cache update**
  - It is inefficient to bring in new version of file (may be large)
  - Re-fetch the file only when the file is needed

Handling disconnected operation

- **Problem: how to extend caching well to cope with disconnection?**
  - Updates to the server are synchronous. If disconnection, it usually leads to timeout and failure in normal distributed file systems
  - While disconnected, it is impossible to bring in new files or updates from server

Upon disconnection

- **Handle updates:**
  - Coda switches to the *disconnected mode*, which is transparent to user
  - Venus emulates the server, and logs updates locally in the *Client Modification Log* (CML) without existence of server.
- **Handle missing files:**
  - Coda uses hoarding to let users keep important files in their hoard database
  - e.g., you hoard calendar and appointment list in a PDA; web pages for disconnected access in browser; files in a laptop that you think will be useful
  - Venus generates temporary file ids and service requests to hoarded files.

Upon reconnection

- **Enters the *data reintegration state* - Venus integrates locally changed files to servers**
- **Coda supports a weaker semantics than session semantics**
  - Does not prevent inconsistent updates but guarantees that inconsistent updates will always be detected
    - Handles only write-write conflicts
    - Apply user conflict resolution

Data reintegration

- **CML is sent to server for consistency checking**
- **If consistent, server installs updates in 4 steps**
  - Initiate a *distributed update transaction* (with one of the server as coordinator) and locks all objects in the log
  - Validate updates on each object and if valid, execute operation
  - Real data for update are then transferred through back-fetching from client
  - Commit the transaction and release all locks
- **If inconsistent, a conflict occurs and must be resolved.**

Data reintegration

- **If another client has modified the same file, there is a *conflict* (write/write conflict)**
- **Principles adopted in Coda:**
  - No update should be lost without explicit user approval.
  - Conflicts are application-specific concept
  - Final decision should be made by human users
- **Conflict resolution in Coda:**
  - Coda uses *last storeid* to indicate the last update to an object
  - Conflict resolution can be based on concurrent versioning systems (using version number, timestamp, version vector).
  - Usually, resolution can be done automatically (e.g., each user updates different part of a file) through the use of Application-specific Resolver (ASP)
  - Sometimes, human intervention is need