Distributed Systems
8L for Part IB

Additional Material
(Case Studies)

Dr. Steven Hand

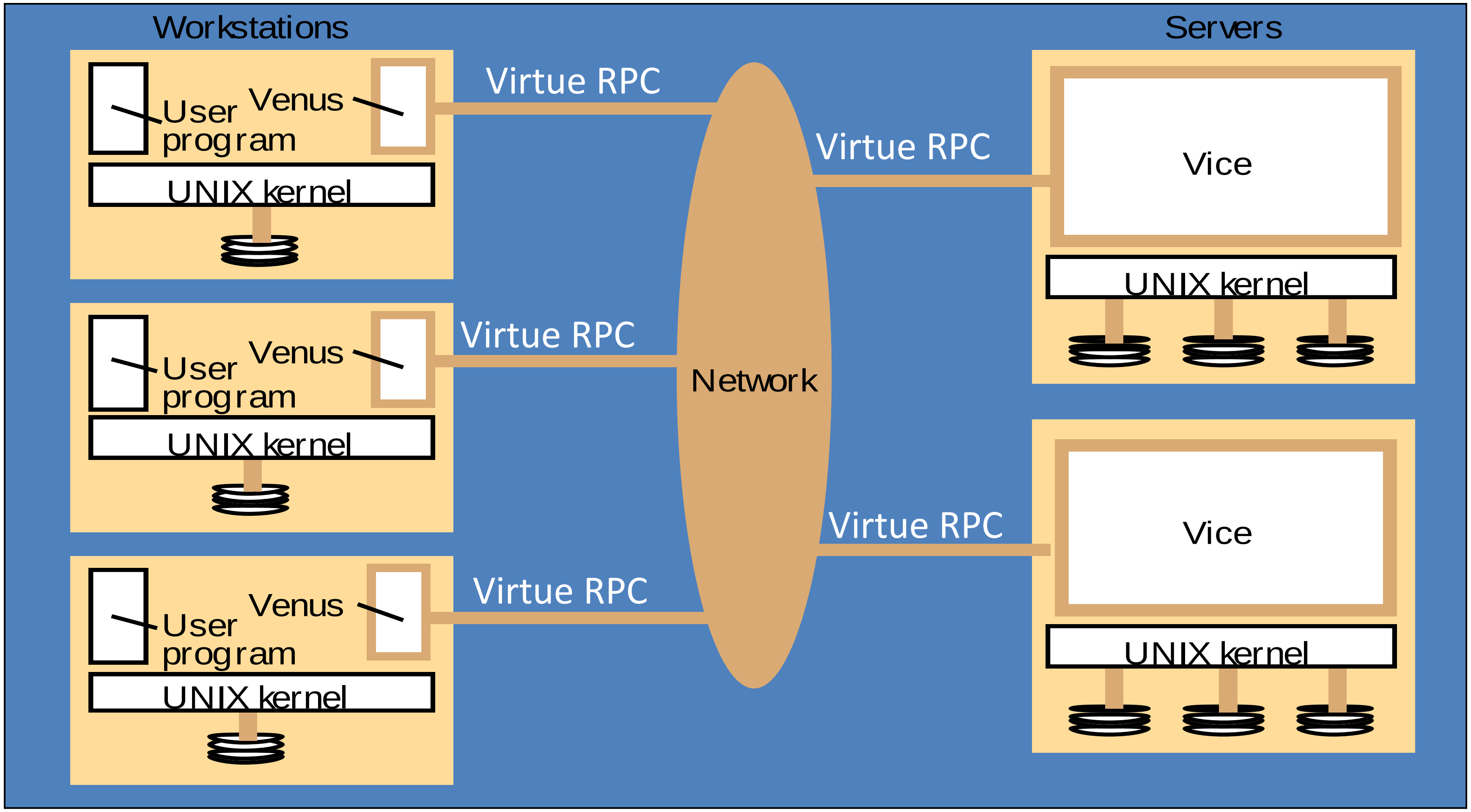
Introduction

- The Distributed Systems course covers a wide range of topics in a variety of areas
- This handout includes 3 case studies (all of which are distributed storage systems):
  - AFS, Coda & Bayou
- Aim is to illustrate how entire systems:
  - use replication,
  - maintain consistency, and
  - handle failure
- The hope is that these will aid understand of core concepts: details covered are **not examinable**

The Andrew File System (1983)

- A different approach to remote file access
- Meant to service a large organization
  - Scaling is a major goal
- Basic AFS model:
  - Files are stored permanently at file server machines
  - Users work from workstation machines
    - With their own private namespace
  - Andrew provides mechanisms to cache user’s files from shared namespace
- Even “local” accesses go via client

Vice, Virtue and Venus…



Basic Idea: Whole File Caching

- Andrew caches *entire files* from the system.
  - On open Venus caches files from Vice
  - On close, [modified] copies are witten back
- Reading and writing bytes of a file are done on the cached copy by the local kernel
- Venus also caches contents of directories and symbolic links, for path-name translation
  - Exceptions for modifications to made directly on the server responsible for that directory

Why do Whole-File Caching?

- Minimizes communications with server
  - Less network traffic
  - Better performance
- Most files used in entirety anyway (prefetch)
- Simpler cache management
- However does requires substantial free disk space on workstations
  - Can be an issue for huge files
  - Later versions allow caching part of a file

Andrew Shared Namespace

- An AFS installation has a global shared namespace
- A **fid** identifies a Vice file or directory
- A fid is 96 bits long and has three 32-bit components:
  - **volume number** (a unit holding the files of a single client)
  - **vnode number** (basically an inode for a single volume)
  - **uniquifier** (generation number for vnodes numbers, thereby keeping certain data structures, compact
- High degree of name and location transparency
  - Fids do not embed any notion of location
  - Every server stores volume->server mapping

AFS Consistency

- Aiming to provide “local” semantics
- Implemented by callbacks:
  - On open, Venus checks if client already has copy
  - If not, then requests from Vice server that is custodian of that particular file
  - Server returns contents along with a *callback promise* (and logs this to durable storage)
- Whenever a client sends back an updated copy (e.g. on close), invoke all callbacks
- Same scheme used for volume map

AFS Pros and Cons (1)

- Performance
  - Most file operations are done locally (and most files typically have one writer in a time window)
  - Little load on servers beyond open/close timescales
- Location transparency
  - Indirection via volume map makes it easy to move volumes
  - Also can do limited replication (read-only files)
- Scalability
  - Initial design aimed for 200:1 client-server ratio
  - Indirection and caching makes this easily achievable
- "Single System Image"
  - Clients (workstations) essentially interchangeable

AFS Pros and Cons (2)

- Good Security
  - Client machines untrusted (only Vice servers trusted)
  - Strong initial authentication via Kerberos
  - Can use encryption used to protect transmissions
- But:
  - Complex and invasive to install (“take over the world”)
  - Usability issues, e.g. ticket expiration, weird “last close wins” semantics for concurrent update
- Ultimately AFS popular only in niche domains…

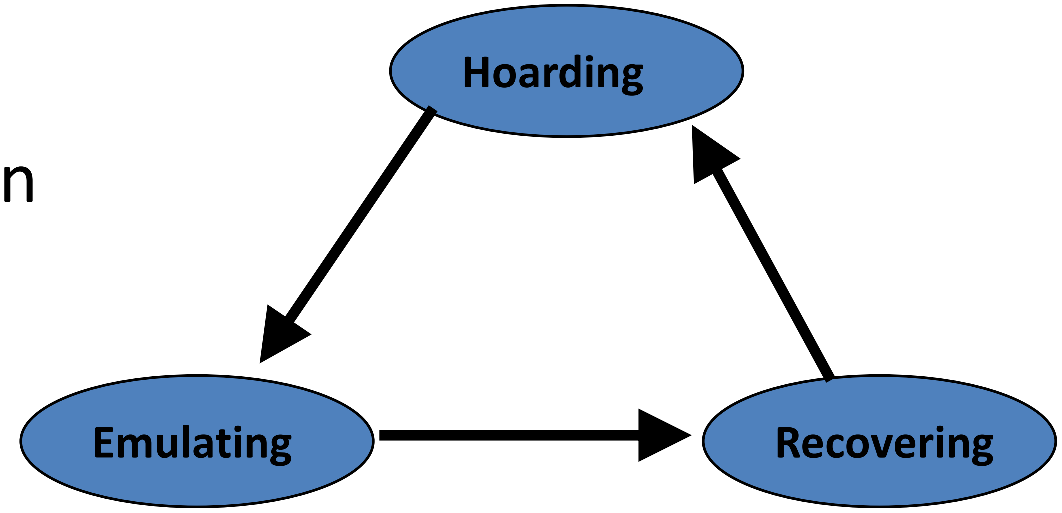
Coda (CMU, 1987+)

- A file-system with **optimistic replication**
  - Essentially client/server (developed from AFS)
- Motivated by the emergence of laptops
  - When connected to network, laptop operated just like any other Andrew workstation
  - When disconnected, however, no file updates allowed once the leases expired
  - This was fine for temporary outages in AFS (e.g. reboot or network glitch), but not for mobile use

Coda Operation

- Change the Venus cache manager to operate in three different modes:

1. Hoarding
   - "Normal" operation
2. Emulating
   - Disconnected
3. Reintegrating
   - Reconciling changes back to the server



- Few changes required to Vice or Virtue

Coda: Hoarding

- "Normal" operation a little different than AFS
  - Aggressively cache copies of files on local disk
- Add a Hoard Database (HDB) to Coda clients
  - Specifies files to be cached on local disk
  - User can tweak HDB, and add priorities
    - Laptop disks were <u>small</u> back in the day
  - Files actually cached a function of hoard priority and actual usage – can pickup dependencies
- Perform *hoard walk* periodically (or on request) to ensure disk has only highest priority files

Coda: Emulating

- When disconnected, attempts to access files not in the cache appear as failures to apps
- All changes made to anything are written in a persistent log (the client modification log)
  - In implementation was managed by using lightweight recoverable virtual memory (lRVM)
  - Simplifies Venus itself
- Venus purges unnecessary entries from the CML (e.g. updates to files later deleted)

Coda: Reintegrating

- Once a coda client is reconnected, it initiates a reintegration process
  - Performed one volume at a time
  - Venue ships replay to each volume
  - Volumes execute a log replay algorithm
  - Basic conflict detection and ‘resolution’
- Lessons learned:
  - Reintegration can take a long time (need fast network)
  - Conflicts rare in practice (0.75% chance of update of same file by two users within 24 hours)

Coda: Summary

- Generally better than AFS
  - Inherits most AFS advantages, but adds more
  - e.g. replicated Vice servers with writable replicas
  - e.g. CML can end up coalescing updates (or removing them entirely) => less traffic, server load
- Much simpler than peer-to-peer schemes:
  - Client only needs to reconcile with "its" server
  - Servers themselves strongly connected + robust
  - Garbage collection straightforward

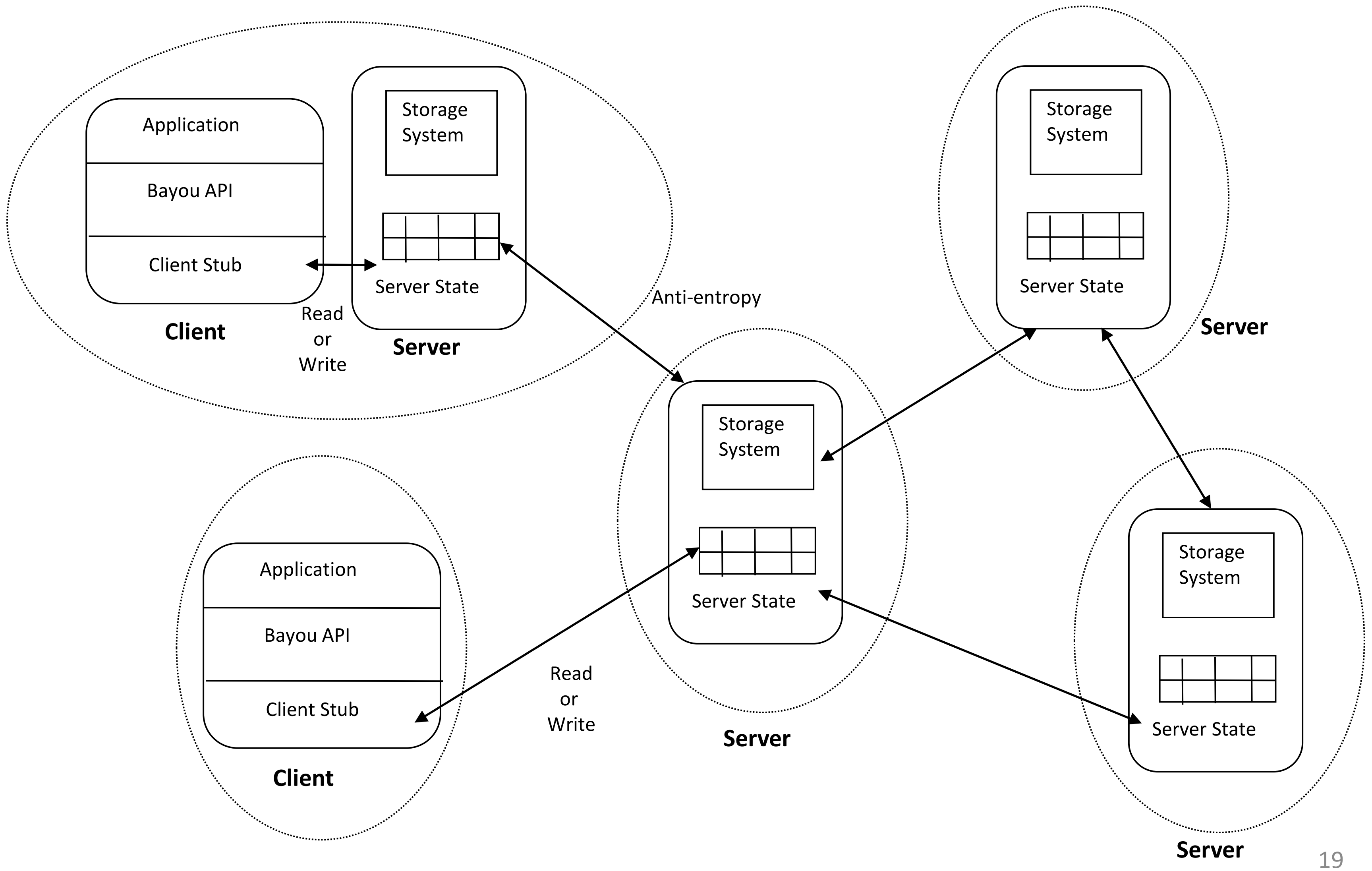
Bayou (Xerox PARC, 1994)

- Bayou is a ***weakly consistent*** distributed storage system
  - Assumes a set of nodes (e.g. laptops, PDAs) which are only occasionally connected
  - Clients talk to servers, each of which manages a ‘database’ (= set of data items)
  - (clients and servers can be co-located on a node)
  - Any client is allowed to read from or write to any server with which it can communicate
- Built around idea of <u>eventual consistency</u>

Eventual Consistency

- Assumption that, if no more updates, all servers will end up with the same data
- Relies on two underlying properties:
  - Total propagation: every server eventually receives every update (perhaps via intermediaries)
  - Consistent ordering: every server can agree on the order of all (non-commutative) updates
- To achieve these Bayou relies on ***anti-entropy***, logical ordering, and a commit protocol

The Bayou Architecture



Bayou Contributions

- Three key contributions:
  - Definition of various *session guarantees*
  - Design and implementation of database, anti-entropy scheme, and means to achieve total ordering
  - Application-specific conflict management
- Session guarantees
  - Clients can get confused by lack of read/write ordering in a weakly replicated system
  - Global ("one-copy") ordering at odds with weak consistency – but what if just focus on one session?

Guarantees for sessions

- Basic operations are generalized reads and writes
  - Read: database ‘query’, or just a request for a given file
  - Write: creating, modifying, or deleting items
- Guarantees from single client session point of view
  - Which may span multiple servers
- E.g. Read your writes (RYW):
  - Reads should reflect previous session writes
  - e.g. Password change in Grapevine system
- “Guarantees” in that data system will either ensure them for each read/write, or notify that it can’t

Read Your Writes

- Want to ensure that if a session writes something, a subsequent read sees it
- Imagine each write at a server is assigned a globally unique id (WID), e.g. <server, seq>
- Servers maintain DB(S): ordered list of all WIDs
- Client remembers its session *write set*
  - List of WIDs for all of its writes
- Then can provide RYW by:
  - Before doing a read, client asks server S for DB(S)
  - Only continue if write set if a subset of DB(S)

Monotonic Reads

- Ensure that every session read sees a non-decreasing set of writes
  - I.e. don't want to operate on older version of data
  - E.g. client requests list of new email messages, and then requests full body of one of them
- To achieve this, on every read the server returns the *relevant writes*: minimal subset of WIDs
- Client adds this to its session *read set*
- Then can provide monotonic reads by:
  - Before doing read from server S, ask for DB(S)
  - Only proceed if read set is a subset of DB(S)

Writes Follow Reads

- Ensures traditional write after read dependencies apply at all servers
  – If a session has a write W following a read(s), then W will be ordered at every server after any writes whose effects were seen by that read(s)
- Affects users outside the session:
  – Essentially says casual order within session must become the overall order
- Needs a bit more work than previous two…

Ensuring WFR

- If a server S accepts a write W at time t, it must ensure W > W* for any W* in DB(S,t)
  - i.e. new writes must be ordered after known ones
- When anti-entropy is performed between S and S2, then any W in DB(S,t) can only be propagated if all W* < W in DB(S,t) are also
  - i.e. must transfer complete history
- If these two hold, can implement WFR as:
  - On any session read, add relevant writes to read-set
  - Before writing to S, check read set is a subset of DB(S)

Monotonic Writes

- Essentially provides traditional write ordering
  - i.e. if W1 precedes W2 in a session then, for any S, if W2 is in DB(S) then W1 is too, and W1<W2
- Relies on the same two additional properties as writes follow reads
- With these, ensuring monotonic writes means:
  - Client maintains write set
  - Before writing to S, ask for DB(S)
  - Only continue if write set is a subset of DB(S)

Session Guarantees: Summary

- Four independent (but related) properties

	Session State Updated	Session State Checked
Read Your Writes	Write	Read
Monotonic Reads	Read	Read
Writes Follow Reads	Read	Write
Monotonic Writes	Write	Write

- First two require little support
- Second two require: (a) new writes ordered after existing ones, and (b) propagation is inclusive

Bayou Implementation

- Inspired by session guarantee model, but made more practical
  - Model suggests servers maintain log of updates for ever which leads to massive read/write sets
- Basic idea: at each server, consider writes to be in one of two states:
  - Tentative Writes, and
  - Stable (or Committed) Writes
- Stable writes are those who have been globally agreed upon – hence server can "collapse" these into resulting data and forget about WIDs etc.

Tentative and Committed Writes

- When a write is accepted by a server, it is initially deemed *tentative*
  - Given WID of <server, timestamp>
- Timestamps in Bayou are logical clocks:
  - Initially synchronized to local system clock
  - However if receive message (e.g. during anti-entropy) with larger clock, warp clock forward
- At some stage, writes become *committed*
  - All committed writes are ordered before any tentative ones, i.e. $[\ W_1 < W_2 < ..\ W_N\ ] < [\ TW_1 < TW_2 < TW_3\ ]$

Committing Writes

- For W to be committed, everyone must agree on:
  - Total order of all previous committed entries
  - Fact that W is next in total order
  - Fact that all uncommitted entries are “after” W
- Bayou does this via a primary commit protocol
- “Primary” replica marks each write it receives with permanent CSN (commit sequence number)
  - That write is now committed
- Nodes exchange CSNs via gossip
  - CSNs define total order for committed writes

Why Primary Commit?

- Chosen to match expected environment
  - Many servers may be unavailable at any time
  - Primary commit means that progress (in terms of commitment) can be made with just one node
  - Can have different primaries for different subsets
- In partitioned network may end up with servers having lots of tentative writes
  - That's what session guarantees are for!
  - When network heals, commit order becomes known
  - "recent" tentative writes may be ordered quite "late"

Write Log Management

- Each server maintains partitioned log
  - Front has (subset of) committed writes
  - Back has tentative writes in arrival order
  - Log entries used during anti-entropy
- Also has tuple store for serving reads
  - Represents view of world for reads
  - Tuples have 2-bit tag { committed , tentative }
- Also has undo log for every tentative entry
  - So can recompute “world” if reordering occurs

Anti-Entropy Algorithm

- Each server maintains vector $V[S_1, S_2, \ldots, S_N]$
  - Entry V[S] contains highest sequence number of update received from server S
- Maintain prefix property:
  - If get W from S2, also get all W* < W in DB(S2)
- Algorithm for sever S1 to update server S2:
  - Get S2's version vector V2
  - For each write W in S1's write log, let R=server[W]
    - If V2[R] < W, update S2 with W* <= W for W* in DB(R)

Bayou: Dependency Checks

- So far have talked about reads and writes in very general way
- In practice, this can lead to problems with conflicts, e.g. consider calendar application
  - Update W1 adds booking for Room 5 at 10-11
  - Update W2 adds booking for Room 5 at 11-12
  - Stupid conflict detection might think there's a conflict since both updates affect same tables
- Bayou proposed novel way to handle this…

Make “Writes” into Programs

- Rather that just submitting writes, submit a short program in a scripting language
  - includes a *dependency check*, and a *merge procedure*
- E.g. { Book meeting Room 5 10-11 if free; otherwise book Room 8 12-1; else fail }
- The dependency check here is “if free”
  - Essentially a query on the state of the database
- “Merge procedure” here is just try another room
  - But can be considerably more sophisticated depending on application.. see SOSP paper for details

Bayou: Summary

- Weak consistency for high availability
- Recognized that clients can get confused by arbitrary ordering: session guarantees
- (Some assumptions on servers required)
- Also argued that applications on such systems must be aware of weaker semantics
  - Need to handle notion of tentative writes
  - Can use programmatic updates for ‘atomicity’