Distributed File Systems

Distributed file system goals

- **Access transparency**
  - Clients unaware files are remote
- **Location transparency**
  - Consistent name space (local and remote)
- **Concurrency transparency**
  - Modifications are coherent
- **Failure transparency**
  - Client and client programs should operate correctly after server failure
- **Heterogeneity**
  - File service should be provided across different hardware and software platforms

Distributed file system goals

- **Scalability**
  - Scale from a few machines to many (tens of thousands?)
- **Replication transparency**
  - Clients unaware of replication
  - Coherence maintained
- **Migration transparency**
  - Files should be able to move around without clients' knowledge
- **Fine grained distribution of data**
  - Locate objects near processes that use them

File service types

Remote access model

File service provides functional interface:

- *create, delete, read bytes, write bytes, etc…*

Advantages:

- Client gets only what's needed
- Server can manage coherent view of file system

Problem:

- Possible server and network congestion
    - Servers are accessed for duration of file access
    - Same data may be requested repeatedly

File server

- **File Directory Service**
  - Maps textual names for file to internal locations that can be used by file service
- **File service**
  - Provides file access interface to clients
- **Client module** (driver)
  - Client side interface for file and directory service
  - if done right, helps provide access transparency

    e.g. under vnode layer

Naming issues

Should all machines have the exact same view of the directory hierarchy?

- e.g., global root directory?

```
//server/path
/remote/server/path
```

or….

Should each machine have its own hierarchy with remote resources located as needed?

```
/usr/local/games
```

Location transparency

Is the name of the server known to the client?

- //server1/dir/file
- Server can move without client caring
- If file moves to server2 … problems!

Location independence

- Files can be moved without changing the pathname

Access transparency

- Allow applications to access remote files as local files
- Remote FS name space should be syntactically consistent with local name space
  1. redefine the way all files are named and provide a syntax for specifying remote files
     - e.g. //server/dir/file
     - Can cause legacy applications to fail
  2. use a file system *mounting* mechanism
     - Overlay portions of another FS name space over local name space

Semantics of
file sharing

Absolute time ordering

Sequential semantics

Read returns result of last write

Easily achieved *if*

- Only one server
- Clients do not cache data

BUT

- Performance problems if no cache
  - Obsolete data
- We can ***write-through***
  - Must notify clients holding copies
  - Requires extra state, generates extra traffic

Paul Krzyzanowski • Distributed Systems

Session semantics

- Relax the rules
- Changes to an open file are initially visible only to the process (or machine) that modified it.

System design issues

Name resolution (*namei*)

(a) Component at a time

vs.

(b) entire path at once

(b) is more efficient but…

- Remote server may access and reveal more if its file system than it wants
- Other components cannot be mounted underneath remote tree

Can use *(a)* and cache bindings

Stateful or stateless?

Stateful

- Server maintains client-specific state

- Shorter requests
- Better performance in processing requests
- Cache coherence is possible
  - Server can know who's accessing what
- File locking is possible

Stateful or stateless

Stateless

- Server maintains *no* information on client accesses

- Each request must identify file and offsets
- Server can crash and recover
  - No state to lose
- Client can crash and recover
- No open/close needed
  - They only establish state
- No server space used for state
  - Don't worry about supporting many clients
- Problems if file is deleted on server
- File locking not possible

Paul Krzyzanowski • Distributed Systems

Caching

Hide latency to improve performance for repeated accesses

Four places

- Server's disk
- Server's buffer cache
- Client's buffer cache
- Client's disk

WARNING: cache consistency problems

Approaches to caching

- **Write-through**
  - What if another client reads its cached copy?
  - All accesses will require checking with server
  - Or Server maintains state and sends invalidations
- **Delayed writes**
  - Data can be buffered locally (consistency suffers)
  - Remote files updated periodically
  - One bulk wire is more efficient than lots of little writes
  - Problem: semantics become ambiguous

Approaches to caching

- Write on close
  - Admit that we have session semantics

- Centralized control
  - Keep track of who has what open on each node
  - Stateful file system with signaling traffic

Distributed File Systems
Case Studies

NFS

Network File System

Sun Microsystems

c. 1985

NFS Design Goals

- Any machine can be a **client** or **server**
- Must support **diskless workstations**
- **Heterogeneous systems** must be supported
  - Different HW, OS, underlying file system
- **Access transparency**
  - Remote files accessed as local files through normal file system calls (via VFS in UNIX)
- **Recovery from failure**
  - Stateless, UDP, client retries
- **High Performance**
  - use caching and read-ahead

NFS Design Goals

No migration transparency

If resource moves to another server, client must remount resource.

NFS Design Goals

No support for UNIX file access semantics

- Stateless design: file locking is a problem.
- All UNIX file system controls may not be available.

NFS Design Goals

Transport Protocol

Initially NFS ran over **UDP** using Sun RPC

Why UDP?

Slightly faster than TCP

No connection to maintain (or lose)

Designed for ethernet LAN environment

relatively reliable

Error detection but no correction.

NFS retries requests

NFS Protocols

- Mounting protocol
  - Request access to exported directory tree
- Directory & File access protocol
  - Access files and directories (read, write, …)

Mounting Protocol

- Send pathname to server
- Request permission to access contents

client:	parses pathname contacts server for file handle

- Server returns **file handle**
  - File device #, inode #, instance #

client:	create in-code vnode at mount point. (points to inode for local files) points to **rnode** for remote files - *stores state on client*

Mounting Protocol

- **static mounting**
  - Mount request contacts server

Server: `/etc/exports`

Client: `mount fluffy:/users/paul /home/paul`

Directory and file access protocol

- Initially perform ***lookup*** RPC
  - returns **file handle** and attributes

- *Not* like *open*
  - No information is stored on server
- handle passed as a parameter for other file access functions
  - e.g. `read(handle, offset, count)`

Directory and file access protocol

- NFS has 16 functions
  - (version 2; six more added in version 3)

null
lookup

create
remove
rename

read
write

link
symlink
readlink

mkdir
rmdir
readdir

getattr
setattr

statfs

Accessing files

- Parse **component at a time** via *namei*
  - At each point, see if mount point
    - Yes? Continue on the mounted file system
    - Remote? Perform NFS RPC *lookup*
- Ensures that **..** is processed locally and future mount points are processed
- Final lookup returns handle
- Create in-core vnode, rnode

Accessing files

Application can now access file

file descriptor → in-core vnode (VFS layer)

in-core rnode (NFS client)

Perform NFS read/write RPCs using state in rnode

RPCs include user ID and group ID

- security hole

NFS Performance

- Usually slower than local
- Improve by caching at client
  - Goal: reduce number of remote operations
  - Cache results of
    read, readlink, getattr, lookup, readdir
  - Cache file data at client (buffer cache)
  - Cache file attribute information at client
  - Cache pathname bindings for faster lookups
- Server side
  - Caching is "automatic" via buffer cache
  - All NFS writes are *write-through* to avoid unexpected data loss if server dies

Inconsistencies may arise

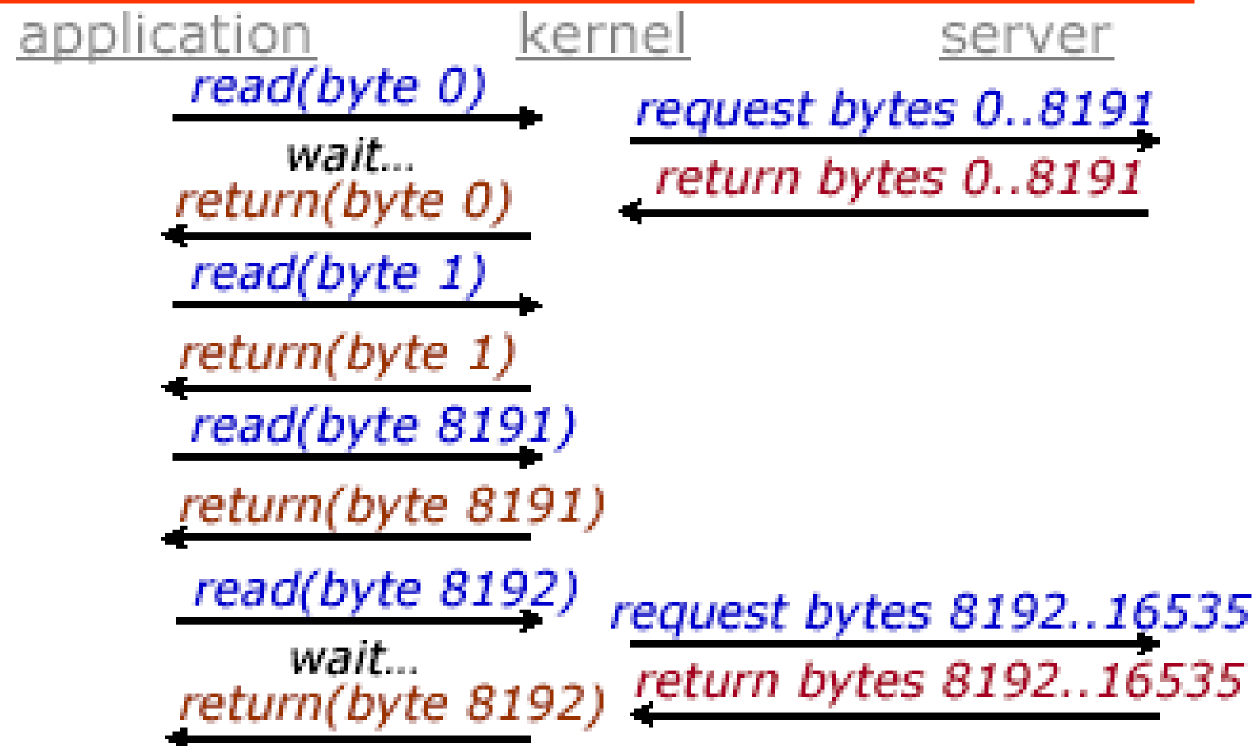
- Try to resolve by **validation**
  - Save timestamp of file
  - When file opened or server contacted for new block
    - Compare last modification time
    - If remote is more recent, invalidate cached data

Validation

- Always invalidate data after some time
  - After 3 seconds for open files (data blocks)
  - After 30 seconds for directories
- If block is modified
  - Marked *dirty*
  - Scheduled to be written
  - Flushed on close

NFS read-ahead

- Transfer data in large chunks
  - 8K bytes default
- As soon as a chunk is received
  - A new read request is issued for the next chunk
  - Assumes data is read in-order


NFS read-ahead
application
kernel
server
read(byte 0)
request bytes 0..8191
wait...
return bytes 0..8191
return(byte 0)
read(byte 1)
return(byte 1)
read(byte 8191)
return(byte 8191)
read(byte 8192)
request bytes 8192..16535
wait...
return bytes 8192..16535
return(byte 8192)
Paul Krzyzanowski • Distributed Systems

Problems with NFS

- File consistency
- Assumes clocks are synchronized
- Open with append cannot be guaranteed to work
- Locking cannot work
  - Separate lock manager added (stateful)
- No reference counting of open files
  - You can delete a file you (or others) have open!
- Global UID space assumed

Problems with NFS

- No reference counting of open files
  - You can delete a file you (or others) have open!
- Common practice
  - Create temp file, delete it, continue access
  - Sun's hack:
    - If same process with open file tries to delete it
    - Move to temp name
    - Delete on close

Problems with NFS

- File permissions may change
  - Invalidating access to file
- No encryption
  - Requests via unencrypted RPC
  - Authentication methods available
    - Diffie-Hellman, Kerberos, Unix-style
  - Rely on user-level software to encrypt

More improvements… NFS v3

- New version of NFS protocol
- Support 64-bit file sizes
- TCP support and large-block transfers
  - UDP caused more problems on WANs (errors)
  - All traffic can be multiplexed on one connection
    - Minimizes connection setup
  - No fixed limit on amount of data that can be transferred between client and server
- Server checks access for entire path from client

More improvements… NFS v3

- Negotiate for optimal transfer size
- New *commit* operation
  - Check with server after a *write* operation to see if data is committed
  - If *commit* fails, client must **resend** data
  - Reduce number of *write* requests to server
  - Speeds up *write* requests
    - Don't require server to write to disk immediately
- Return file attributes with each request
  - Saves extra RPCs

SMB

Server Message Blocks

Microsoft

c. 1987

SMB Goals

- File sharing protocol for Windows 95/98/NT/2000/ME/XP
- Protocol for sharing
  - Files, devices, communication abstractions (named pipes), mailboxes
- Servers: make file system and other resources available to clients
- Clients: access shared file systems, printers, etc. from servers

Design Priority:

locking and consistency over client caching

SMB Design

- Request-response protocol
  - Send and receive ***message blocks***
    - name from old DOS system call structure
  - Send *request* to server (machine with resource)
  - Server sends response
- Connection-oriented protocol
- Each message contains:
  - Fixed-size header
  - Command string (based on message) or reply string

Message Block

- Header: [fixed size]
  - Protocol ID
  - Command code (0..FF)
  - Error class, error code
  - Tree ID – unique ID for resource in use by client (handle)
  - Caller process ID
  - User ID
  - Multiplex ID (to route requests in a process)
- Command: [variable size]
  - Param count, params, #bytes data, data

SMB Commands

- Files
  - Get disk attr
  - create/delete directories
  - search for file(s)
  - create/delete/rename file
  - lock/unlock file area
  - open/commit/close file
  - get/set file attributes

SMB Commands

- Print-related
  - Open/close spool file
  - write to spool
  - Query print queue
- User-related
  - Discover home system for user
  - Send message to user
  - Broadcast to all users
  - Receive messages

Protocol Steps

- Establish connection
- Negotiate protocol
- Authenticate/set session parameters
  - Send ***sesssetupX*** SMB with username, password
  - Receive NACK or UID of logged-on user
  - UID must be submitted in future requests

Protocol Steps

- Establish connection
- Negotiate protocol - *negprot*
- Authenticate - *sesssetupX*
- Make a connection to a resource
  - Send *tcon* (tree connect) SMB with name of shared resource
  - Server responds with a **tree ID** (TID) that the client will use in future requests for the resource

Protocol Steps

- Establish connection
- Negotiate protocol - *negprot*
- Authenticate - *sesssetupX*
- Make a connection to a resource – *tcon*
- Send open/read/write/close/… SMBs

Locating Services

- Clients can be configured to know about servers
- Each server broadcasts info about its presence
  - Clients listen for broadcast
  - Build list of servers
- Fine on a LAN environment
  - Does not scale to WANs
  - Microsoft introduced *browse servers* and the *Windows Internet Name Service* (WINS)

Security

- Share level
  - Protection per "share" (resource)
  - Each share can have password
  - Client needs password to access all files in share
  - Only security model in early versions
  - Default in Windows 95/98
- User level
  - protection applied to individual files in each share based on access rights
  - Client must login to server and be authenticated
  - Client gets a UID which must be presented for future accesses

SMB evolves

SMB reverse-engineered

- **samba** under Linux

Microsoft released protocol to X/Open in 1992

Microsoft, Compaq, SCO, others joined to develop an enhanced public version of the SMB protocol:

Common Internet File System

(CIFS)

Goals

- Heterogeneous HW/OS to request file services over network
- Based on SMB protocol
- Support
  - Shared files
  - Byte-range locking
  - Coherent caching
  - Change notification
  - Replicated storage
  - Unicode file names

Goals

- Applications can register to be notified when file or directory contents are modified
- Replicated virtual volumes
  - For load sharing
  - Appear as one volume server to client
  - Components can be moved to different servers without name change
  - Use *referrals*
  - Similar to AFS

Goals

- Batch multiple requests to minimize round-trip latencies
  - Support wide-area networks
- Transport independent
  - But need reliable connection-oriented message stream transport
- DFS support (compatibility)

Caching and Server Communication

- Increase effective performance with
  - Caching
    - Safe if multiple clients reading, nobody writing
  - read-ahead
    - Safe if multiple clients reading, nobody writing
  - write-behind
    - Safe if only one client is accessing file
- Minimize times client informs server of changes

Oplocks

Server grants **opportunistic locks** (**oplocks**) to client

- Oplock tells client how/if it may cache data
- Enhancement of DFS tokens

Client must request an oplock

- oplock may be
  - Granted
  - Revoked
  - Changed by server

Level 1 oplock

- Client can open file for exclusive access
- Arbitrary caching
- Cache lock information
- Read-ahead
- Write-behind

If another client opens the file, the server has former client **break its oplock**:

- Client must send server any lock and write data and acknowledge that it does not have the lock
- Purge any read-aheads

Level 2 oplock

- Request if expect others to read
- Multiple clients may have the same file open as long as none are writing
- Cache reads, file attributes
- Send other requests to server

Level 2 oplock revoked if another client opens the file for writing

Batch oplock

- Client can keep file open on server even if a local process that was using it has closed the file
- Client requests batch oplock if it expects programs may behave in a way that generates a lot of traffic (e.g. accessing the same files over and over)
  - Designed for Windows batch files

Batch oplock revoked if another client opens the file

Filter oplock

- Open file for read or write
- Locks file so other clients cannot open for write or delete
  - All clients can share read access
- Allow other clients to perform non-intrusive (read) operations

No oplock

- All requests must be sent to the server

- can work from cache *only* if byte range was locked by client

CIFS Summary

- Standard has not yet materialized
  - Future uncertain
- Oplocks mechanism supported in Windows NT, 2000, XP
- Oplocks offer flexible control for distributed consistency

http://www.pk.org/rutgers/notes/pdf/dsm-talk.pdf

Distributed Shared Memory

Motivation

- SMP systems
  - Run parts of a program in parallel
  - Share single address space
    - Share data in that space
  - Use threads for parallelism
  - Use synchronization primitives to prevent race conditions
- Can we achieve this model with multicomputers?
  - All communication and synchronization must be done with messages

DSM

- Goal: allow networked computers to share memory

- How do you make a distributed memory system appear local?
- Physical memory on each node used to hold pages of shared virtual address space

Take advantage of the MMU

- Page table entry for a page is valid if the page is held (cached) locally
- Attempt to access non-local page leads to a **page fault**
- **Page fault handler**
  - Invokes DSM protocol to handle fault
  - Fault handler brings page from remote node
- Operations are transparent to programmer
  - DSM looks like any other virtual memory system

Simplest design

- Each page of virtual address space exists on only *one* machine at a time (no caching)

Simplest design

On page fault:

- Consult central server to find which machine is currently holding the page
- **Directory**

Request the page from the current owner

- Current owner invalidates PTE
- Sends page contents
- Recipient allocates frame, reads page, sets PTE
- Informs directory of new location

Problem

- Directory becomes a bottleneck
  - All page query requests must go to this server
- Solution
  - **Distributed directory**
  - Distribute among all processors
  - Each node responsible for portion of address space
  - Find responsible processor:
    - *hash(page#)mod processors*

Design Considerations: granularity

- Memory blocks are typically a multiple of a node's page size
  - To integrate with a VM system
- Large pages are good
  - Cost of migration amortized over many localized accesses
- BUT
  - Increases chances that multiple objects reside in one page
    - Thrashing
    - **False sharing**

Design Considerations: replication

- What if we allow copies of shared pages on multiple nodes?
- Replication (caching) reduces average cost of read operations
  - Simultaneous reads can be executed locally across hosts
- Write operations become more expensive
  - Cached copies need to be invalidated or updated
- Worthwhile if reads/writes is high

Replication

- Multiple readers, single writer
  - One host can be granted a r/w copy
  - **Or** multiple hosts granted read-only copies

Replication

- Read operation:
  - If block not local
    - Acquire read-only copy of the block
    - Set access writes to read-only on any writeable copy on other nodes
- Write operation:
  - If block not local or no write permission
    - Revoke write permission from other writable copy (if exists)
    - Get copy of block from owner (if needed)
    - Invalidate all copies of block at other nodes

Full replication

- Extend model
  - Multiple hosts have read/write access
  - Need **multiple-readers, multiple-writers protocol**
  - Access to shared data must be controlled to maintain consistency

Dealing with replication

- Keep track of copies of the page
  - Directory with single node per page not enough
  - Maintain **copyset**
    - Set of all systems that requested copies
- Request for page copy
  - Add requestor to copyset
  - Send page contents
- Request to invalidate page
  - Issue invalidation requests to all nodes in copyset and wait for acknowledgements

Consistency Model

- Definition of when modifications to data may be seen at a given processor

- Defines **how memory will appear to a programmer**
  - Places restrictions on what values can be returned by a *read* of a memory location

Consistency Model

- Must be well-understood
  - Determines how a programmer reasons about the correctness of a program
  - Determines what hardware and compiler optimizations may take place

Sequential Semantics

- Provided by most (uniprocessor) programming languages/systems
- **Program order**

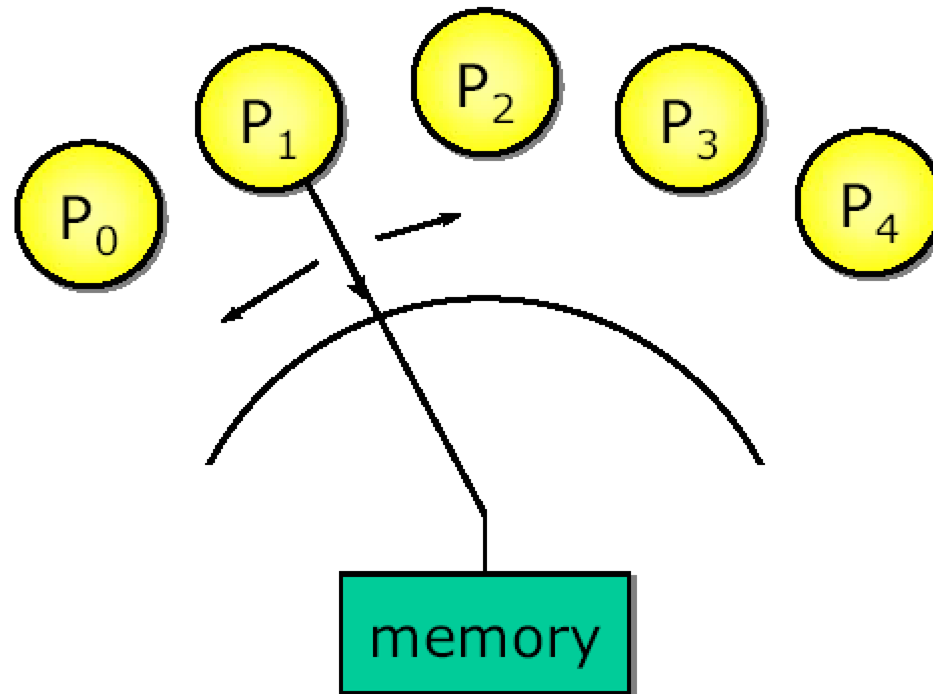
The result of any execution is the same as if the operations of all processors were executed in <u>*some*</u> *sequential order* ***and*** *the operations of each individual processor appear in this sequence in the order specified by the program.*

— Lamport

Sequential Semantics

- Requirements
  - All memory operations must execute one at a time
  - All operations of a single processor appear to execute in program order
  - Interleaving among processors is OK

Sequential semantics



Achieving sequential semantics

- Illusion is efficiently supported in uniprocessor systems
  - Execute operations in program order when they are to the same location or when one controls the execution of another
  - Otherwise, compiler or hardware can reorder
- Compiler:
  - Register allocation, code motion, loop transformation, …
- Hardware:
  - Pipelining, multiple issue, …

Achieving sequential consistency

- Processor must ensure that the previous memory operation is complete before proceeding with the next one
  - **Program order requirement**
  - Determining completion of write operations
    - get acknowledgement from memory system
  - If caching used
    - Write operation must invalidate or update messages to all cached copies.
    - ALL these messages must be acknowledged

Achieving sequential consistency

- Writes to the same location must be visible in the same order by all processes
  - **Write atomicity requirement**
  - Value of a *write* will not be returned by a *read* until all updates/invalidates are acknowledged
    - hold off on read requests until write is complete

Improving performance

- Break rules to achieve better performance
  - Compiler and/or programmer should know what's going on!

- Relaxing sequential consistency
  - Weak consistency

Relaxed (weak) consistency

- Relax program order between all operations to memory
  - Read/writes to different memory operations can be reordered
- Consider
  - Operation in critical section (shared)
  - One process reading/writing
  - Nobody else accessing until process leaves critical section
- No need to propagate writes sequentially *or at all* until process leaves critical section

Synchronization variable

- Operation for synchronizing memory
- All local writes get propagated
- All remote writes are brought in to the local processor
- Block until memory synchronized

Consistency guarantee

- Access to synchronization variables are sequentially consistent
  - All processes see them in the same order
- No access to a synchronization variable can be performed until all previous writes have completed
- No read or write permitted until all previous accesses to synchronization variables are performed
  - Memory is updated

Problems with weak consistency

- Inefficiency
  - Synchronization
    - Because process finished memory accesses or is about to start?
- Systems must make sure
  - All locally-initiated writes have completed
  - All remote writes have been acquired

Can we do better?

- Separate synchronization into two stages:
  - 1. **acquire access**
    - Obtain valid copies of pages
  - 2. **release access**
    - Send invalidations for shared pages that were modified locally to nodes that have copies.

acquire(R) // start of critical section
Do stuff
release(R) // end of critical section

Eager Release Consistency (ERC)

Let's get lazy

- Release requires
  - Sending invalidations to copyset nodes
  - **And** waiting for all to acknowledge
- Delay this process
- On *release*:
  - Send invalidation only to directory
- On *acquire*:
  - Check with directory to see whether it needs a new copy
    - Chances are not every node will need to do an acquire
- Reduces message traffic on releases

Lazy Release Consistency (LRC)

Finer granularity

- Release consistency
  - Synchronizes *all* data
  - No relation between lock and data
- Use **object granularity** instead of **page granularity**
  - Each variable or group of variables can have a synchronization variable
  - Propagate only writes performed in those sections
  - Cannot rely on OS and MMU anymore
    - Need smart compilers

Entry Consistency

How do you propagate changes?

- Send entire page
  - Easiest, but may be a lot of data
- Send differences
  - Local system must save original and compute differences

Home-based algorithms

- Home-based
  - A node (usually first writer) is chosen to be the **home** of the page
  - On *write*, a non-home node will send changes to the home node.
    - Other cached copies invalidated
  - On *read*, a non-home node will get changes (or page) from home node
- Non-home-based
  - Node will always contact the directory to find the current owner (latest copy) and obtain page from there

Home-based Lazy Release Consistency

- At *release*
  - Diffs are computed
  - Sent to owner (home node)
- Home node:
  - Applies diffs as soon as they arrive
- At *acquire*
  - Node requests page from the home node