

Lecture 3

Communication Mechanisms

Plan for today

- ◆ What is needed from a communication mechanism
- ◆ Various communication mechanisms
 - Remote “fork”
 - Distributed shared memory
 - Remote procedure call
 - Data streams
 - Basic message passing
- ◆ Advanced issues
 - Load balancing
 - Efficiency of transport
 - Service location and binding

Functions of the communication mech.

- ◆ Invocation of remote services
 - Initiating an execution on a remote system
- ◆ Inter-process communication
 - Communicating with processes (possibly on remote system)
- ◆ Data movement
 - Moving data between processes (possibly across systems)
- ◆ Sub-goals
 - Hiding distribution issues to simplify programming
 - Balancing load across systems auto-magically
 - Minimizing the performance cost of inter-machine comm.

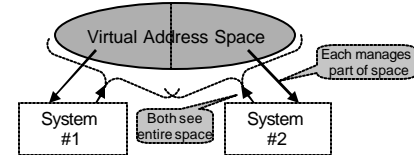
Remote Fork

- ◆ “fork” a program on a remote system
 - just to get a program running on other machine
 - think of it exactly the same as running a program locally
- ◆ Several implementation options
 - possibly send arguments
 - possibly receive results (e.g., exit code)
 - possibly receive output (e.g., stdout)
 - possibly wait for completion
- ◆ Examples
 - r* utilities: rcp, rsh, rexec, ...
 - inetd: accepts connections and forks processes on demand

Distributed Shared Memory (DSM)

- ◆ A common virtual address space shared across systems
 - A given virtual page is the same actual data
 - Obviously, there is work involved in making this happen
- ◆ What is the point?
 - Distributed programming becomes threaded programming
 - Many other aspects of distribution hidden by DSM
- ◆ Base implementation
 - Use VM hardware to point at right physical page, if local
 - or to invoke the DSM fault handling code
 - Move actual data pages to local physical memory on fault
 - fetching it from whoever has most up-to-date copy
- ◆ Various implementation details dictate performance

An example DSM implementation



- ◆ Each system manages a fraction of shared VA space
 - that is, track latest copy and cache consistency info
 - the others fault in and cache pages (ala client file caching)
 - consistency rules are necessarily strict (to simplify apps)
- ◆ More advanced systems may
 - migrate page ownership and management
 - track sub-pages to minimize false sharing

Remote procedure call

- ◆ Procedure calls
 - clean entry, wait, and return from one bit of code to another
 - compiler/system has clear semantics for passing parameters and results (though global variables and such cloud things)
- ◆ System calls
 - procedure calls that cross from application to kernel
 - stub routines convert compiler rules to OS interface rules
 - stubs in the kernel setup tracking state and possibly copyin data
- ◆ RPC: remote procedure call
 - procedure calls that cross from one "system" to another
 - stub routines convert compiler rules into well-defined cross-system generic form
 - receiving stub routines convert generic form to specific local form

Implementation of RPC

- ◆ Automatic stub generation
 - from description of procedure name, parameter and return value types, and generic form rules
 - reduces the busy-work of constructing send/receive stubs
- ◆ Receiver-side invocation
 - caller calls, waits, and restarts upon return of control
 - this is normal procedure call semantics
 - callee starts, executes, and ends
 - this is a little weird, but makes sense given context
- ◆ Pointer de-referencing and bulk data movement
 - pointed-to data must be copied
 - and new pointers constructed at receiver side
 - need support for bulk data movement or continuous streams
 - for example, consider file descriptors or read() buffers

Data streams

- ◆ Data pushed at consumers in steady stream
 - as opposed to having it be requested incrementally
 - generally, an initial request sets up the data stream
- ◆ Benefits
 - Greater server efficiency (not constantly queried by consumers)
 - Better parallelism between producer & consumer
 - Simplifies load balancing (see “Cluster I/O With River”)
- ◆ Limitations
 - Only really works for very regular applications
 - Consumer must be able to describe what to send
 - Must have enough data to transfer to amortize setup overhead

Random message passing

- ◆ Messages sent from one process to another as needed
 - No restrictions and no waiting required
 - Most general form of communication mechanism
 - with this flexibility comes a lot of complexity (in form of gotchas)
- ◆ How to do it
 - every process/system has queue for incoming messages
 - simply send messages as needed to other queues
- ◆ Issues
 - synchronization requires waiting for specific messages
 - but other messages may come in first, requiring extra buffering
 - queues can fill up
 - potential for deadlock as each process blocks in sending messages
 - kernel decides who runs when
 - so message processing may be delayed

Load balancing

- ◆ When systems give “work orders” to other systems
 - whether client/server requests or other schemes
- ◆ Proactive schemes
 - coordinated server selection
 - either single work producing or intermediary work distributor
 - decentralized server selection
 - each using round-robin, random, observation-based
- ◆ Reactive schemes
 - load shedding
 - overloaded servers ask others to handle some load
 - work stealing
 - idle servers ask others for stuff to do

Load balancing cont.

- ◆ Restrictions: can't just send work orders anywhere
 - Only producers with the desired sources can provide data
 - Data destined to particular locations must go there
 - Often only specific systems can handle a given work order
- ◆ Special cases
 - Streams of work orders
 - easier to balance without centralization, since feedback is plentiful
 - Data production from sensors or storage
 - Allowing load balancing requires replication

Transport Protocols

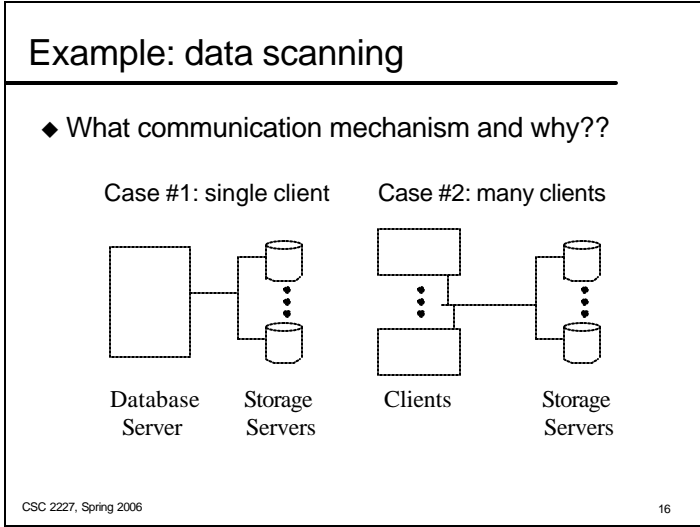
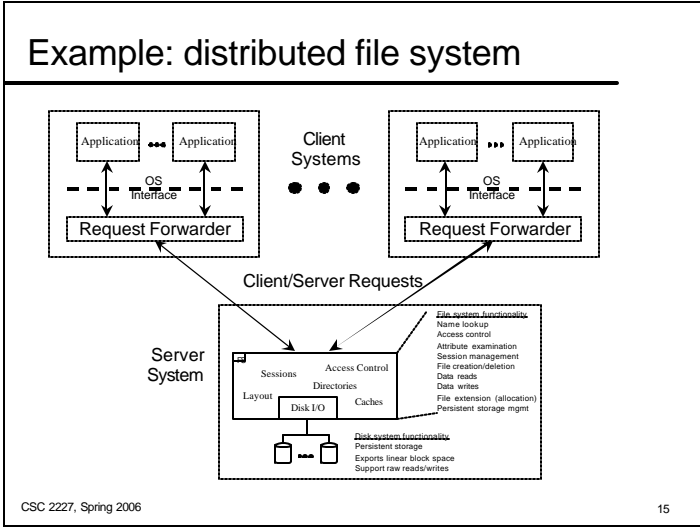
- ◆ Specialized (a la Birrell)
 - tried to minimize bytes transferred
 - no longer really an issue
 - assumed single-packet call and single-packet return
 - single "call #" allows simple transmit and acknowledgement scheme
 - main limitation is dealing with bulk data movement
- ◆ General UDP-based (a la DCE RPC)
 - issues of retransmit and flow control must be handled
 - use of "pipes" for bulk data
 - main problem has been complexity of implementation
- ◆ General TCP-based (a la HTTP)
 - handles various networking issues (and then some)
 - main concern (traditionally) has been performance

CSC 2227, Spring 2006 13

Service Location and Binding

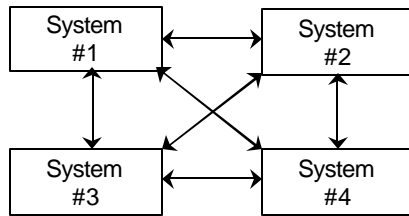
- ◆ Finding and linking up to services
 - machine it runs on and port it listens to
- ◆ Basic options for each
 - hard-coded in clients (often standardized, as in telnet)
 - embedded in server name (e.g., URL) or data ID (e.g., email)
 - ask a name server
- ◆ Name server (aka directory service)
 - Translates service names to location info
 - Can provide one (e.g., DNS) or all (e.g., grapevine) locations
 - Using or giving ability to select server to talk to
 - Main challenge: staying up to date and recovering from oops
- ◆ Binding to services
 - Sort of like TCP: listen, connect, accept, converse

CSC 2227, Spring 2006 14



Example: parallel matrix computation

- ◆ What communication mechanism and why??



Resources

- ◆ Papers for next week's discussion
 - Implementing Remote Procedure Call
 - Birrell & Nelson
 - TreadMarks : Shared Memory Computing on Networks of Workstations
 - Amza et al.
- ◆ Other communication models
 - Cluster I/O with River: Making the fast case common
 - Arpaci-Dusseau et al.