Processes in Distributed Systems

Today

- Threads in distributed systems
- Virtualization
- Thin-client computing
- Servers
- Code migration

Processes and threads

- Processes offer concurrency transparency, but at a relatively high price on performance
- Threads offer concurrency without much less transparency
  - Applications with better performance that are harder to code/debug
- Advantages of multithreading
  - No need to block with every system call
  - Easy to exploit available parallelism in multiprocessors
  - Cheaper communication between components than with IPC
  - Better fit for most complex applications
- Alternative ways to provide threads
  - User-, kernel-level threads, LWP and scheduler activations

Threads in distributed systems – clients

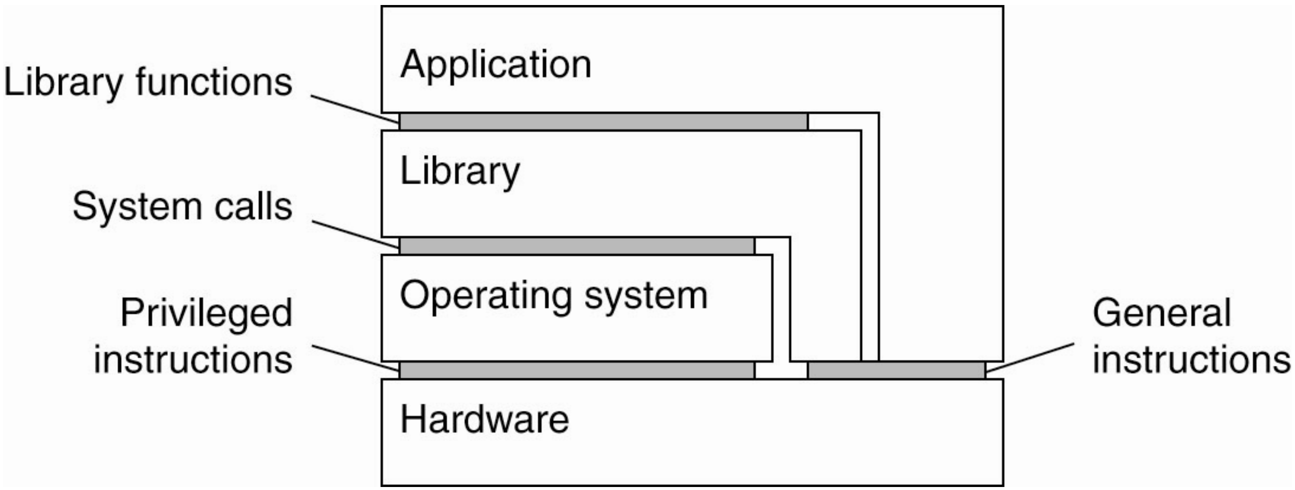
- Client usage is mainly to hide network latency
- E.g. multithreaded web client:
  - Web browser scans an incoming HTML page, and finds that more files need to be fetched
  - Each file is fetched by a separate thread, each doing a (blocking) HTTP request
  - As files come in, the browser displays them
- Multiple request-response calls to other machines:
  - A client does several RPC calls at the same time, each one by a different thread
  - It then waits until all results have been returned
  - Note: if calls are to different servers, we may have a linear speed-up compared to doing calls one after the other

Threads in distributed systems – servers

- In servers, the main issue is improved performance and better structure
- Improve performance:
  - Starting a thread to handle an incoming request is much cheaper than starting a new process
  - Having a single-threaded server prohibits simply scaling the server to a multiprocessor system
  - As with clients: hide network latency by reacting to next request while previous one is being replied
- Better structure:
  - Most servers have high I/O demands. Using simple, well-understood blocking calls simplifies the overall structure.
  - Multithreaded programs tend to be smaller and easier to understand due to simplified flow of control

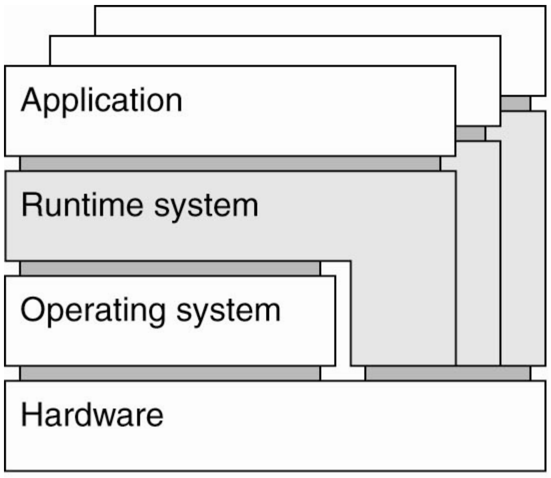
Virtualization

- Virtualization is becoming increasingly important:
  - Hardware changes faster than software
  - Ease of portability and code migration
  - Isolation of failing or attacked components
- Virtualization can take place at very different levels, strongly depending on the interfaces as offered by various systems components:

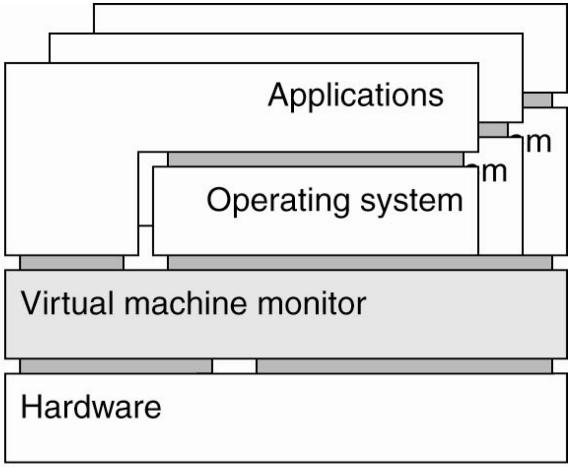


VM architectures

- We should differentiate between process virtual machines and virtual machine monitors:
  a) Process VM: A program compiled to intermediate (portable) code, which is then executed by a runtime system (e.g. Java VM).
  b) VMM: A separate software layer that mimics the instruction set of hardware; a complete operating system and its applications can be supported (e.g.: VMware).



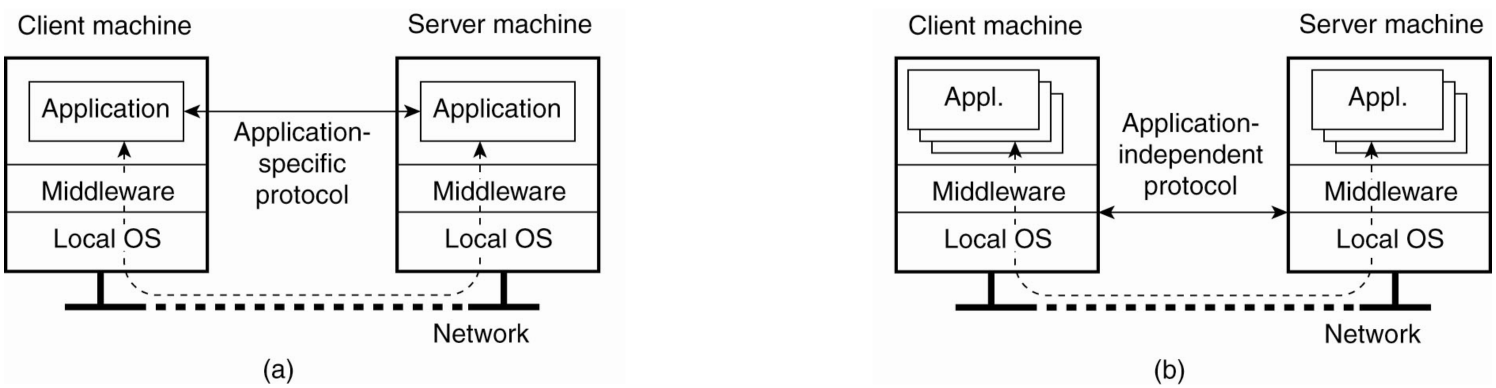
(a)



(b)

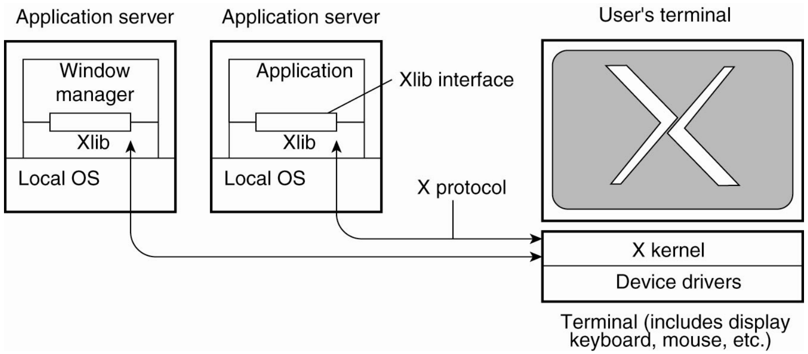
Thin and fat clients

- Client machines provide the means for users to interact with remote servers
  - Fat client – for each remote service, the client machine has a separate counterpart (a)
  - Thin client – client machine is just a terminal providing direct access to remote services (b)



Thin-client network computing

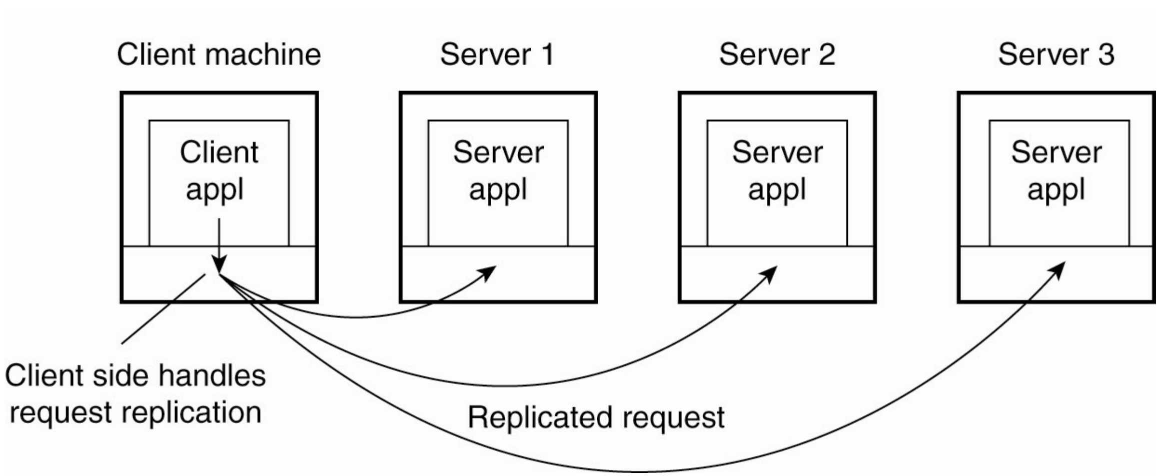
- A major part of client-side software is focused on (graphical) user interfaces.
  - With X, the kernel and the application need not be on the same machine



- Compound documents: User interface is application aware → inter application communication:
  - Drag-and-drop: move objects across the screen to invoke interaction with other applications (trash can)
  - In-place editing: integrate several applications at user-interface level (word processing + drawing facilities)

Client-side software and transparency

- Client-side software is often tailored for distribution transparency
  - Access transparency: client-side stubs for RPCs
  - Location/migration transparency: let client-side software keep track of actual location
  - Replication transparency: multiple invocations handled by client stub
  - Failure transparency: mask server and communication failures



Server design

- Server – a process that waits for incoming service requests at a specific transport address
- Iterative vs. concurrent servers: Iterative servers can handle only one client at a time, in contrast to concurrent servers
- In practice, there is a 1-to-1 mapping between port and service, e.g. ftp: 21, smtp:25
- Superservers: Servers that listen to several ports, i.e., provide several independent services; start a new process to handle new requests (UNIX inetd/xinetd)
  - For services with more permanent traffic get a dedicated server

Out-of-band communication

- How to interrupt a server once it has accepted (or is in the process of accepting) a service request?
- Solution 1: Use a separate port for urgent data (possibly per service request):
  - Server has a separate thread (or process) waiting for incoming urgent messages
  - When urgent msg comes in, associated request is put on hold
    - Require OS supports high-priority scheduling of specific threads or processes
- Solution 2: Use out-of-band communication facilities of the transport layer:
  - E.g. TCP allows to send urgent msgs in the same connection
  - Urgent msgs can be caught using OS signaling techniques

Servers and state

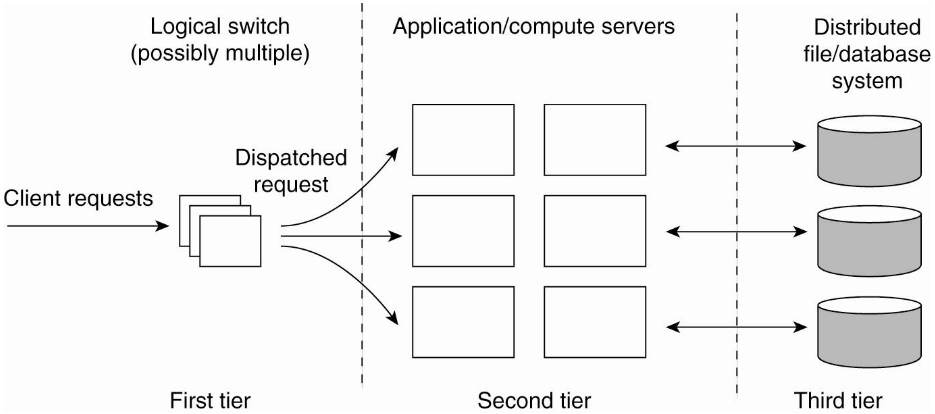
- Stateless servers: Never keep accurate information about the status of a client after having handled a request:
  - Don’t record whether a file has been opened (simply close it again after access)
  - Don’t promise to invalidate a client’s cache
  - Don’t keep track of your clients
- Consequences:
  - Clients and servers are completely independent
  - State inconsistencies due to client or server crashes are reduced
  - Possible loss of performance because, e.g., a server cannot anticipate client behavior (think of prefetching file blocks)

Servers and state

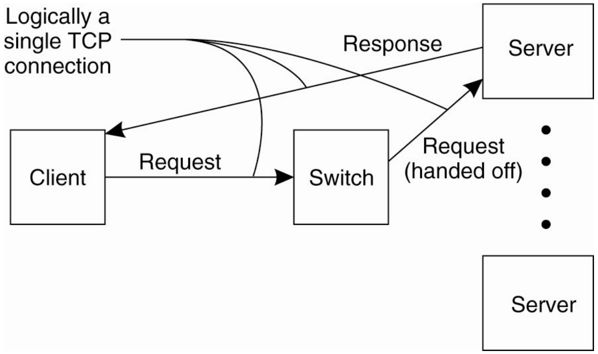
- Stateful servers: Keeps track of the status of its clients:
  - Record that a file has been opened, so that prefetching can be done
  - Knows which data a client has cached, and allows clients to keep local copies of shared data
- Observation: The performance of stateful servers can be extremely high, provided clients are allowed to keep local copies. As it turns out, reliability is not a major problem.

Server clusters

- Many server clusters are organized along three different tiers:

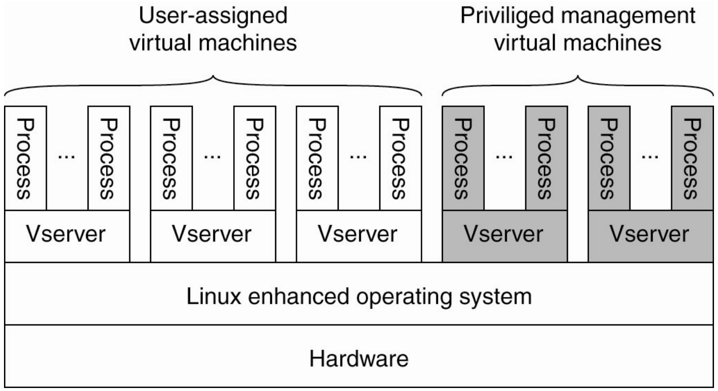


- Key element: The first tier is generally responsible for passing requests to an appropriate server.
  - May lead to a bottleneck.
- Various solutions, but one popular one is TCP-handoff:



Example: PlanetLab

- Different organizations contribute machines, which they subsequently share for various experiments
- Ensure that different distributed applications do not get into each other's way: virtualization:



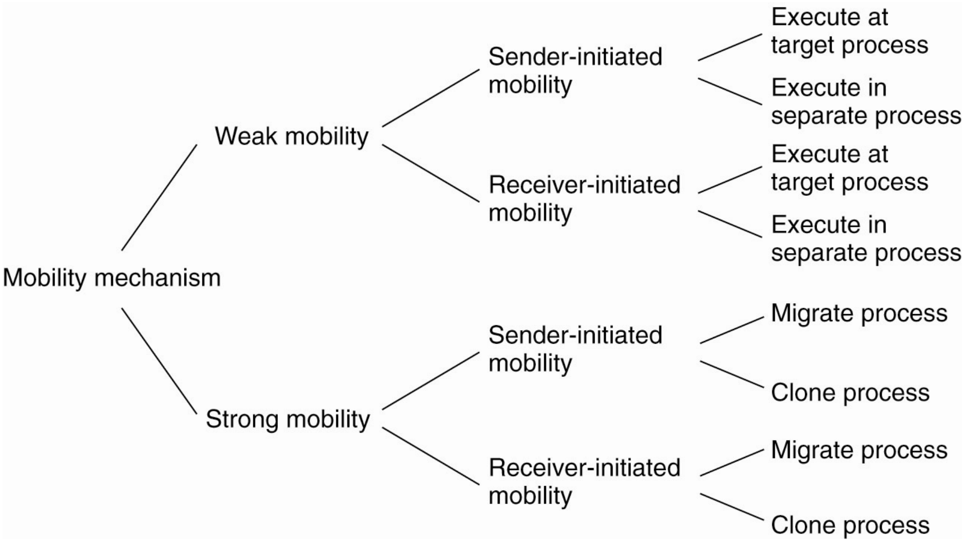
- Vserver: Independent and protected environment with its own libraries, server versions, etc. Applications are assigned a collection of vservers across multiple machines (slice).

Code migration

- Instead of passing data around, why not moving code?
- What for?
  - Improve load distribution in compute-intensive systems
  - Save network resource and response time by moving processing data closer to where the data is
  - Improve parallelism w/o code complexities
    - Mobile agents for web searches
  - Dynamic configuration of distributed systems
    - Instantiation of distributed system on dynamically available resources; binding to service-specific, client-side code at invocation time

Models for code migration

- Process seen as composed of three segments
  - Code segment – set of instructions that make up the program
  - Resource segment – references to external resources needed
  - Execution segment – state of the process (e.g. stack, PC, …)
- Some alternatives
  - Weak/strong mobility – code or code and execution segments
  - Sender or receiver initiated
  - A new process for the migration code?
  - Cloning instead of migration



Migration and local resources

- Process-to-resource binding
  - Binding by identifier – process is bound to a socket
  - Binding by value – need only the value of a resource, e.g. standard library
  - Binding by type – need only a resource of a certain type, e.g. printer
- Resource-to-machine binding
  - Unattached resources – easily moved, e.g. files
  - Fastened resources – costly to move, e.g. large database
  - Fixed resource – tightly bound to a location, e.g. local devices, sockets
- E.g. file, memory page, socket?

Resource-to-machine binding

		Unattached	Fastened	Fixed
Process-to-resource binding	By identifier	MV (or GR)	GR (or MV)	GR
	By value	CP (or MV,GR)	GR (or CP)	GR
	By type	RB (or MV,CP)	RB (or GR,CP)	RB (or GR)

GR Establish a global systemwide reference
MV Move the resource
CP Copy the value of the resource
RB Rebind process to locally-available resource

Migration in heterogeneous environments

- In heterogeneous settings, the target machine may not be OK to execute the migrated code
- The definition of process/thread/processor context is highly dependent on local hardware, OS and runtime system
- Only solution: Make use of an abstract machine that is implemented on different platforms
- Current solutions:
  - Interpreted languages running on a VM (Java/JVM; scripting languages)
  - Virtual machine monitors, allowing migration of complete OS + apps – a form of strong mobility

Summary

- Processes are a fundamental piece of distributed systems – how they are internally organized is key
- The basic client/server organization has a number of interesting details to work with
  - From thin/fat clients to server designs for scalability and easy of management
- Typically one thinks of moving data, but moving processes has a number of interesting advantages and technical complexities
  - Virtual machines may help us deal with quite a few of the technical issues