# Introduction Distributed Systems
# Processes & Threads


## Today

- Processes and Threads
- Clients and servers
- Thin-client computing
- Code migration

# Processes and threads

- Distributed system
  - A collection of independent, interconnected processors …
- Processes – virtual processors, offer concurrency transparency, at a relatively high price on performance
- Threads offer concurrency w/ 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

# 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

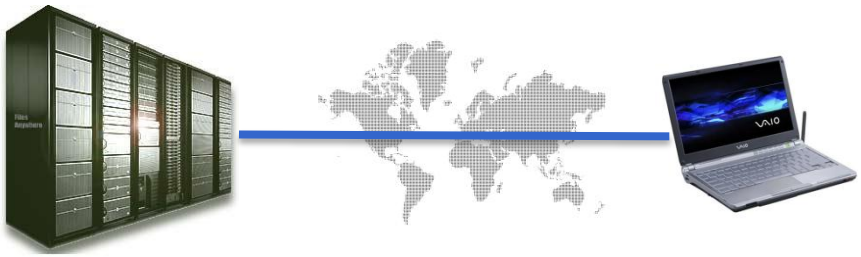
- Stateless servers: Never keep accurate information about the status of a client after having handled a request:
  - Don’t record whether a file has been opened (simply close it again after access)
  - Don’t promise to invalidate a client’s cache
  - Don’t keep track of your clients
- Consequences:
  - 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 pre-fetching 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.

# Thin-client computing

- Thin-client
  - Client and server communicate over a network using a remote display control
    - Client sends user input, server returns screen updates
  - Graphical display can be virtualized and served to a client
  - Application logic is executed on the server



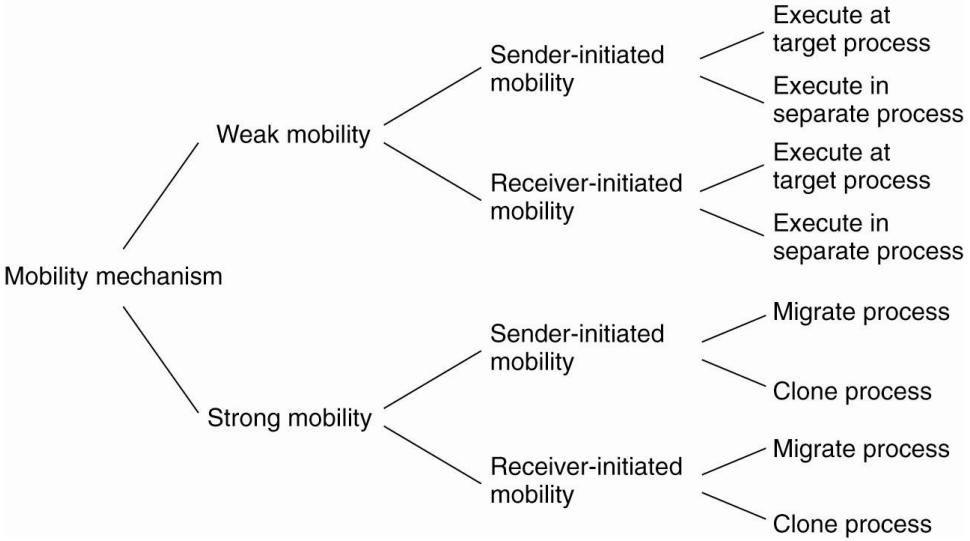
- Technology enablers
  - Improvements in network bandwidth, cost and ubiquity
  - High total cost of ownership for desktop computing
- Big business opportunity
  - Sun Microsystems, Google, Microsoft, AT&T Virtual Network Computing, …

# 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



# Question 3

- *What drives the market for application service providers? What would be the main factor limiting wide-area thin-client performance?*