Clients and Servers (Processing)

March 2, 2017

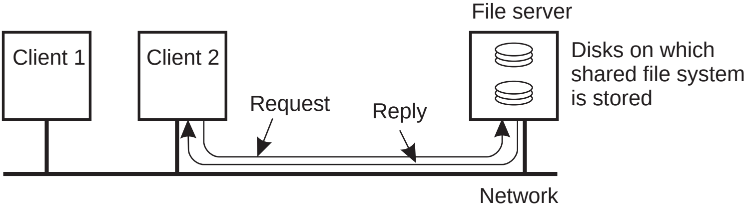
Roadmap

- Definition
- Server/Object Location
- Distribution Transparency
- Concurrency
- Keeping State in Servers
- Failures
- Security
- Communication Channel Adaptation
- Further Reading

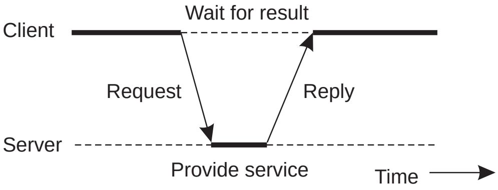
Roadmap

Clientes e Servidores

- Most distributed applications have a **client-server** architecture:



- We'll use *client* and *server* in a broad sense:



- A server can also play the role of client of another service.

Roadmap

Server/Object Location

Problem: how does a client find a server?

Solution: not one, but several alternatives:

- hard coded, rarely;
- program arguments: more flexible, but ...
- configuration file
- via *broadcast*/*multicast*;
- via location/naming server (later in the course)
  - local, like `portmapper` or `rmiregistry`;
  - global.

Roadmap

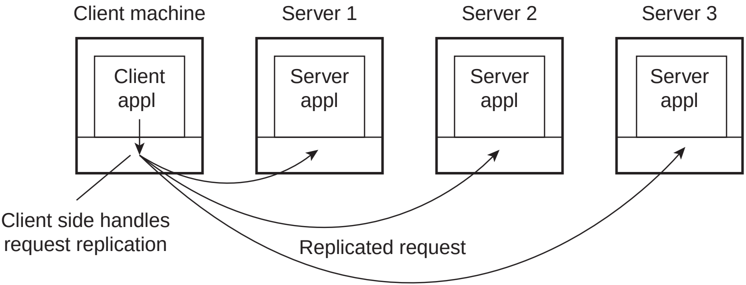
Distribution Transparency

Issue: Many distribution transparency facets can be achieved through client side **stubs** (also called **clerks**):

Acess e.g. via RPC;

Location e.g. via multicast;

Replication e.g. by invoking operations on several replicas:



Faults e.g. by masking server and communication faults

- if possible

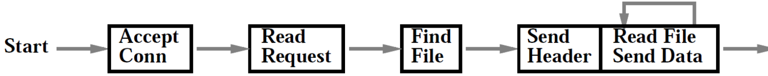
Roadmap

Concurrency

- There are several reasons for using concurrency:
  - Performance (+ on servers);
  - Usability (+ on clients) – still performance, really.
- The goal is to ovelap I/O with processing
- Example: Web service

  Client-side
  - A Web page may be composed of several *objects*
  - A browser can render some objects, while it fetches others via the net.

  Server-side
  - May serve several requests simultaneously



src:Pai et al. 99

How to Achieve Concurrency?

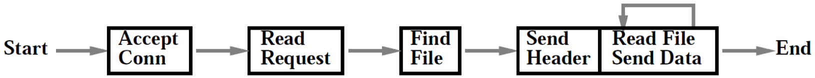
Threads

- Remember SO ...

Events

- Remember LCOM ...

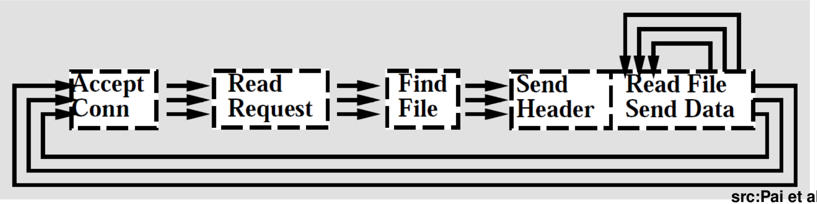
Iterative Web Server



src:Pai et al. 99

- Has only one thread
- Processes a request/connection at a time
- To read the file the server may have to go to disk. In that case, it:
  - will block and
  - cannot process other requests
- Such a server can process only a few requests per time unit

Multi-threaded Server



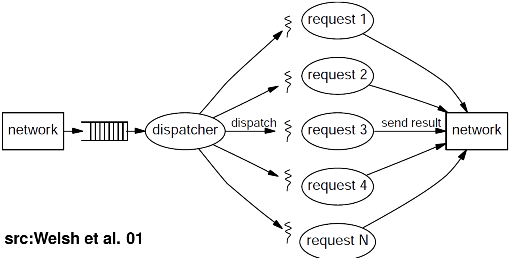
src:Pai et al. 99

- Each thread processes a request (and HTTP 1.0 connection)
- If the number of threads is larger than the number of cores/processors
  - When one thread blocks on I/O
  - Another thread may be scheduled to run in its place.

- A common pattern is:

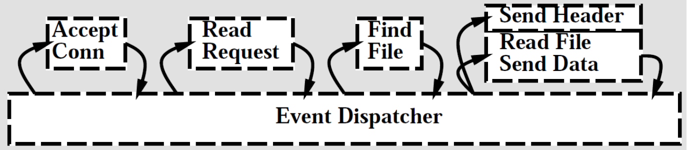
  One dispatcher thread, which accepts the requests

  Several worker threads, which process the requests



src:Welsh et al. 01

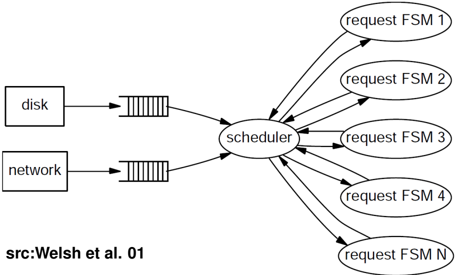
Event-driven Server



src:Pai et al. 99

- The server executes a loop, in which it:
  - waits for events (usuallly I/O events)
  - processes these events (sequentially)
- Blocking is avoided by using **non-blocking** I/O operations
- Known as the state machine approach
  - The state of the server evolves in response to events

- For more complex services:
  - Each request is a FSM
  - The loop dispatches the event to the appropriate FSM



src:Welsh et al. 01

Thread vs. Event Debate

Ease of programming

Performance

Thread-based Concurrency: Ease of Programming

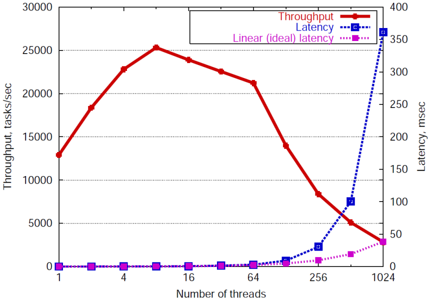
- Appears simple:
  - Structure of each thread similar to that of an iterative server
  - Need **only** to ensure **isolation** in the access to shared data structures
- Could use only monitors and condition variables, e.g. synchronized methods in Java
  - Not so easy: there are some implications in terms of modularity (Ousterhout96)
  - Possibility of deadlocks
- Performance may suffer
  - The larger the critical sections, less concurrency
  - But the main reason for concurrency is performance

Event-based Concurrency: Ease of Programming

- Programmer needs to:
  - Break processing according to potentially blocking calls
  - Manage the state explicitly (using state machines), rather than relying on the stack
- The structure of the code is very different from that of the iterative server
- No nasty errors like race conditions, which may be elusive
- But many complain about lack of support by debugging tools
- ... and others that the it leads to poorly structured code
  - Actually, the author interestingly points out the issue is preemption rather than multithreading

Thread-Based Concurrency: Performance

- Same file 8 KB reads (no disk accesss)
- No thread creation
- "4-way 500MHz Pentium III with 2 GB memory under Linux 2.2.14"



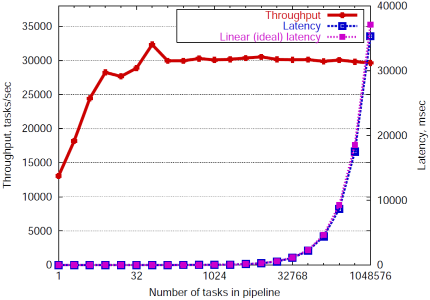
src: Welsh et al. 01

- As the number of threads increases, the system throughput increases, then levels-off and finally dives
- Clearly each thread requires some resources
- There are also issues concerning context switching
  - Actually, depend on whether user-level or kernel-level threads

Event-Based Concurrency: Performance

- Requires non-blocking (also called asynchronous) I/O operations
  - Otherwise, must resort to multiple threads to emulate asynchronous I/O
- Allows user level scheduling
  - The dispatcher may choose which event to handle next

- Same file 8 KB reads (no disk accesss)
- Only one thread
- As the number of requests in a queue increases throughput increases until it reaches a plateau



src: Welsh et al. 01

- Needs multiple threads to achieve **parallelism** in multi core/processor platforms

TB vs EB Concurrency: Performance

- The debate was somewhat "muddled" by implementations that were less than optimal
- Actually, at the technical level this is very similar to the debate about user-level vs. kernel-level threads
- User-level threads are more efficient than kernel-level threads
  - Function calls vs. system calls
  - But efficient implementations require OS support for non-blocking I/O
- But there are some unavoidable blocking, e.g. page faults
- We need kernel-level threads in order to take advantage of multiple processors/cores

Server Architectures

Architecture	Paral.	I/O Oper.	Progr.
Iterative	No	Blocking	easy
Multi-threaded	Yes	Blocking	races
State-machine	Yes	Non-blocking	event-driven

- To take advantage of multiple processors/cores we need to use *kernel-level threads* (or processes).
  - On state-machine designs we may use multiple threads

TB vs EB Concurrency: Conclusion

- Pure thread-based and event-based designs are the extremes in a design space
- Threads are not as heavy as processes, but they still require resources
  - You may want to bound their number
- If you want more parallelism, you need to use an event-based design
- There are many frameworks for supporting event-driven designs
  - Java itself offers Java NIO (non-blocking I/O)
- Not sure about their performance
  - They are often built on top of a stack of multiple layers

Thread-based Concurrency: Practical Considerations

Java

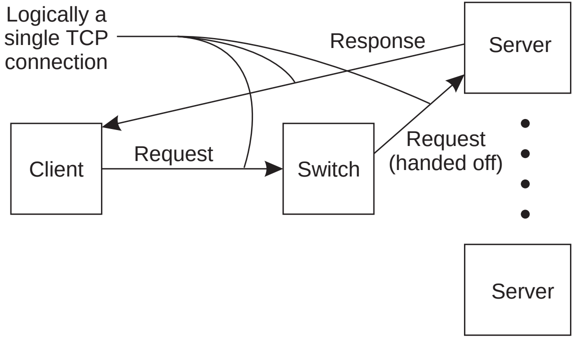
- Assume that the Java socket API is not thread-safe
  - The documentation is mute aboute this
    - Java runs on top of different OS
- You must handle concurrency explicitly

POSIX

- It requires many system calls, such as `accept`, `read/write`, `sendto/receivefrom`, to be **thread-safe**
  - But, data of concurrent `write`'s may be interleaved
    - I.e., `write/read` may not be **atomic** (apparently it depends on the buffer size)
- What about `send(to)/receive(from)`?
  - When used on STREAM sockets, may behave similarly to `write`
  - When used on DATAGRAM sockets, one expects POSIX-atomicity to be implied, but ...
- To be on the safe side, handle concurrency explicitly

Server Clusters

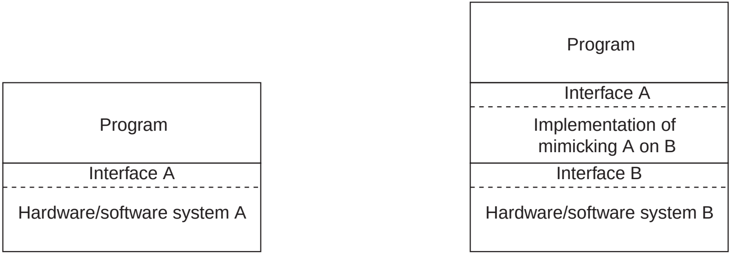
- In order to support Internet-wide services, we need to use **server clusters/server farms.**
- A simple approach is to route the requests at the TCP level:



- The crux is to balance the load on the different servers
  - **Round-robin** is perhaps the simplest approach
  - Application-layer solutions are also possible

Resource Virtualization (1/3)

Idea



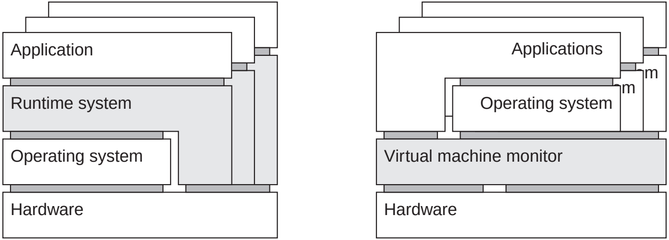
(a) (b)

Essencially, virtualization allows to emulate the behavior of a different system.

Objective Allow legacy SW developed for a specific mainframe architecture to execute on a different platform (IBM)

Resource Virtualization (2/3)

Implementations

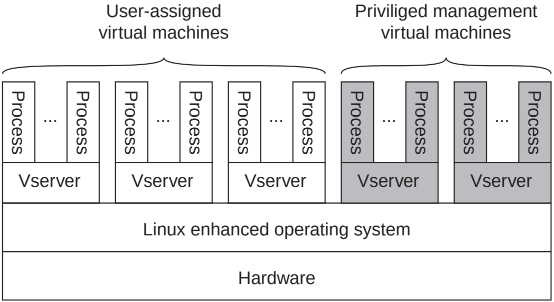


(a) (b)

Process Virtual Machine (a) Each VM provides an instruction set and a run-time to support the execution of a single application. Exemple: *Java Virtual Machine*.

Virtual Machine Monitor (b) Each VM provides an instruction set and a run-time that allows the concurrent and independent execution of multiples OSs. Example: VMware, Xen, VirtualBox.

Resource Virtualization (3/3)



- The use of VMs in distributed systems has mainly two advantages:
  1. Improves security and reliability.
  2. Simplifies management

Roadmap

Servers and State (1/4)

Problem the execution of the same task on every request may unnecessarily tax the server

Solution the server can keep some **state (information)**, i.e. information about the status ongoing interactions with clients;

- the size
- the processing demands

of each message are potentially smaller

- For example, in a distributed file system, the server may avoid open and close a file for each remote read/write operation
  - The server may keep a cache of open files for each client
- Depending on whether or not a server keeps state information, a server is called **stateful** or **stateless**, respectively

Servers and State (2/4)

- Keeping state information raises some challenges:
  - of consistency;
  - of resource management;

  upon failure of either clients or server
- Loss of state when a server crashes may lead to:
  - ignoring or rejecting client requests after recovery:
    - the client will have to start a new **session**
  - wrong interpretation of client requests sent before the crash:
    - TCP connection port reuse
- Keeping state (on server) when the client crashes may lead to:
  - resource depletion
  - wrong interpretation of requests sent by other clients after the crash

Servers and State (3/4)

- But not keeping state information in the server does not solve the problems arising from failures:
  - message duplication may lead to handling the same request several times
    - requests must be **idempotent**, if the transport protocol does not ensure non-duplication of packets;
    - the outcome may not be that satisfactory, if the transport protocol ensures non-duplication of packets

Servers and State(4/4)

- **Obs.-** Statelessness is a protocol issue:
  - A server can be stateless only if each protocol message has all the information for its processing independently of previous communication;
  - Inversely, a server can be stateful only if each protocol message has enough information to relate it to previous communication
- For example, Netscape had to add HTTP-header fields specifically for **cookies**.
  - HTTP is essentially stateless
  - Cookies are a device that allows a server to keep state about a client session:
    - *cookies* are stored on the client side

Client Identification in Stateful Servers

1. Use the address of the **access point**, i.e. of the channel endpoint
   - For example, the client's IP address and port
   - Issue: may not be valid for more than one transport session:
     - E.g. if a TCP connection breaks and a new one is setup in its place, the port number on the client's side may be different
2. Use a transport-layer independent **handle**. For example:
   - HTTP cookies

Roadmap

Failures

Challenges:

1. components in a distributed application may fail, while others continue operating normally
2. on the Internet it is virtually impossible to distinguish network failures from host failures

Solution: highly application dependent

Client Crashes

Challenge resources reserved for the client may remain allocated forever

- sockets, for connection based communication
- state, in the case of stateful servers
- application specific resources

Solution **leases** (and timers):

- a server *leases a resource* to a client for only during a finite time interval: upon its expiration, the resource may be taken away, unless the client **renews** the lease

Server Crashes

Problem I: server may loose state

- may accept duplicated messages after recovery;

Problem II: how can we ensure that a request was performed?

- You are at an ATM. You type your PIN. You choose to withdraw 50 Euros. Suddenly, the machine reports communication error (or was it a server crash?) and does not give you the money. Was it withdrawn from your account?

Roadmap

Security

Challenge: servers execute with priviledges that their clients usually do not have

Solution: servers must

authenticate clients: i.e. "ensure" that a client is who it claims to be;

control access to resources: i.e. "ensure" that a client has the necessary permissions to execute the operation it requests.

- A related requirement is data **confidentiality**
  - need to encrypt data transmitted over the network
- Code migration (i.e. downloaded from the network) raises even more issues.

Roadmap

Communication Channel Adaptation

Order the application will have to reorder the messages (must use a sequence number), if that is important

Reliability need to use timers to recover from message loss. Have to be aware of the possibility of duplicates.

Flow control: if you want to avoid message loss because of insufficient resources

Channel abstraction: the application may have to build messages from a stream. Or, fragment messages at one end and reassemble them at the other end.

Roadmap

Further Reading

- Ch. 3 of Tanenbaum e van Steen, *Distributed Systems, 2nd Ed.*
  - Subsection 3.1.2 *Threads in Distributed Systems*, we assume the remaining material in Section 3.1 to be background knowledge (OS class)
  - Subsection 3.3.2 *Client-Side Software for Distribution Transparency*
  - Section 3.4 *Servers*
  - Section 3.2 *Virtualization*
- Arpaci-Dusseau & Arpaci-Dusseau, *Event-based Concurrency*, Ch. 33 of OSTEP book
- Pai et al., *Flash: An efficient and portable Web Server*, in 1999 Annual Usenix Technical Conference
- Welsh et al, *SEDA: An Architecture for Well-Conditioned, Scalable Internet Services*, in Symposium on Operating Systems, 2001