Clients and Servers (Processing)

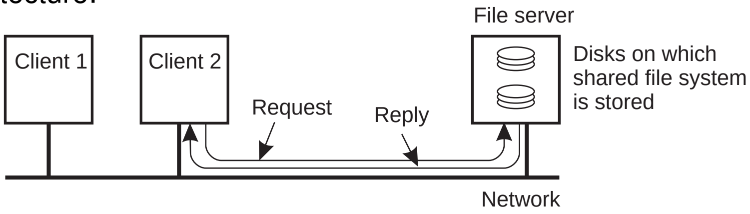
March 3, 2021

Roadmap

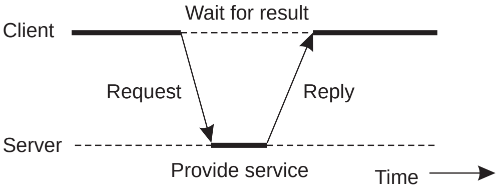
Roadmap

Clients and Servers

- 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

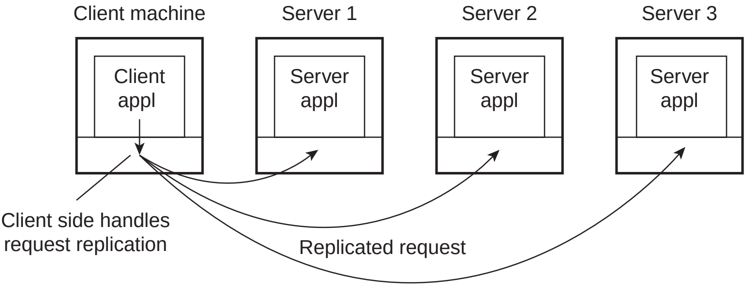
Distribution Transparency

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

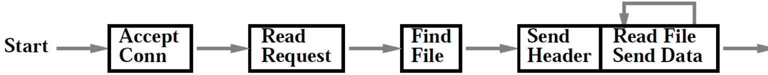
Roadmap

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?

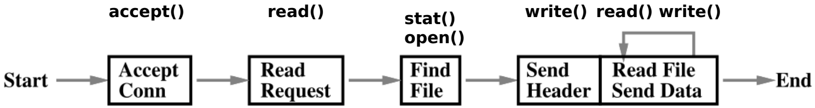
Threads

- Remember SO ...

Events

- Remember LCOM ...

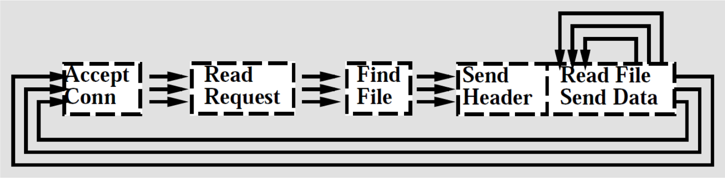
Iterative Web Server



src:Pai et al. 99

- Has only one thread
- Processes a request/connection at a time
- Each step/stage has one operation that can block
  - `stat()` is required because of the HTTP header fields `size` and `last modified`
    - But `open()` may also block
  - Server cannot process other requests while blocked
- Such a server can process only a few requests per time unit

Multi-threaded Server



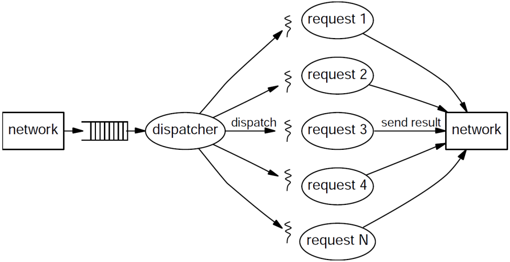
src:Pai et al. 99

- Each thread processes a request (and HTTP 1.0 connection)
- When one thread blocks on I/O
  - Another thread may be scheduled to run in its place.

- A common pattern is:

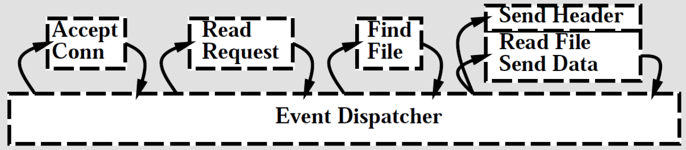
  One dispatcher thread, which accepts a connection request

  Several worker threads, each of which processes all the requests sent in the scope of a single connection



src:Welsh et al. 01

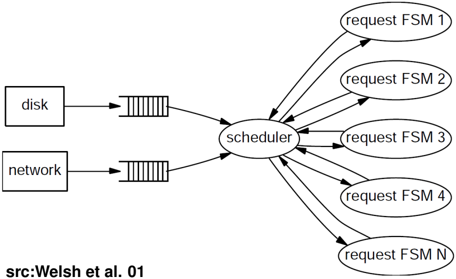
Event-driven Server



src:Pai et al. 99

- The server executes a loop, in which it:
  - waits for events (usuallly I/O events)
  - processes these events (sequentially, but may be not in order)
- Blocking is avoided by using **non-blocking** I/O operations

- Need to keep a FSM for each request
  - The loop dispatches the event to the appropriate FSM
- Known as the state machine approach



src:Welsh et al. 01

Thread vs. Event Debate

Ease of programming

Performance

Thread-based Concurrency: Ease of Programming

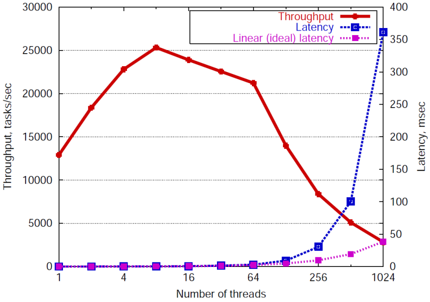
- Appears simple:
  - Structure of each thread similar to that of an iterative server
  - Need **only** to ensure **isolation** in the access to shared data structures
- Could use only monitors, e.g. synchronized methods in Java, and condition variables
  - Not so easy: there are some implications in 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
  - The author points out that the issue is preemption rather than multithreading
  - Actually, the problem is **lack of atomicity**
    - With multiple cores, we can have race conditions, even if there is no preemption

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"



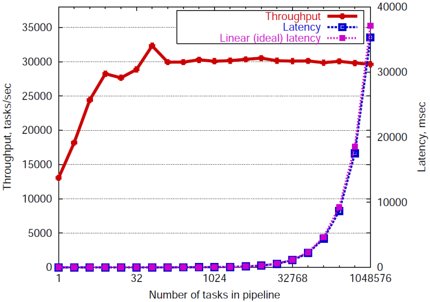
src: Welsh et al. 01

- 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, depends on whether user-level or kernel-level threads

Event-Based Concurrency: Performance

- Requires non-blocking (or asynchronous) I/O operations
  - Otherwise, may use multiple threads for emulation
- 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 performance suffers if OS does not provide non-blocking I/O
  - Worse, there are some unavoidable blocking, e.g. page faults
- 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 both:

  Threads virtually all processors now-a-days are multicore;

  Events to limit the number of threads, and therefore their overhead
- There are many frameworks 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
  - But, often they use thread-based concurrency only by default

Thread-based Concurrency: Basic 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 (C/C++)

- 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

Thread-based Concurrency: Java

Thread class/Runnable interface for creating threads

- You can use also thread pools via the interfaces `java.utils.concurrent.ExecutorService` and/or `java.utils.concurrent.ScheduledExecutorService`

Synchronized methods allow for coarse grained CC, similar to monitors

`java.utils.concurrent.locks` package for synchronization objects (locks and condition variables) to prevent race conds

- Check also the `java.utils.concurrent.Semaphore`
- Some classes of the `java.utils.concurrent` such as `ConcurrentHashMap` provide a thread-safe version of corresponding `java.utils` collection classes

Oracle's Java Tutorials' Concurrency Lesson Overview of core classes

- For a more practical oriented tutorial you can checkout `java.util.concurrent` - Java Concurrency Utilities

Event-based Concurrency with `java.nio` package

Core classes

Channels There are several subclasses

Selector For blocking waiting for more than one I/O event from a selectable channel

Buffers To read/write data from/to channels

Issue `java.nio.channels.FileChannel` is not selectable

- To avoid blockin on file I/O need to use `java.nio.channels.AsynchronousFileChannel`, which supports asynchronous I/O
  - This is more complicated than non-blocking I/O
  - There is no `java.nio.channels.AsynchronousDatagramChannel`, although one can find references to it on the Web

Getting started with new I/O (NIO) Overview of Java I/O

- Refers to non-blocking I/O as asynchronous I/O, but they are not the same
- For (an even) more practical oriented tutorial you can checkout Java NIO Tutorial

Event-driven Server Design by Doug Lea

Doug Lea's design

- This is a presentation :(
- To fill in the details check the Architecture of a Highly Scalable NIO-Based Server Blog

Roadmap

Servers and State

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

Solution the server can keep some **state**, i.e. information about the status of 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
  - Recent cloud-related references, e.g., consider as stateful only if the state is kept in main memory

Stateless File Server

- Consider a simple file service that supports two operations:
  - read data (from file)
  - write data (to file)
- If the server is stateless it keeps no information, therefore each request must include at least:
  - operation
  - client id
  - full path name
  - file offset
  - number of bytes to transfer
  - data (only in write requests)

Upon a read request the server must:

1. Check permissions for client
2. Open the file (`open()`)
3. Set the file offset as requested (`lseek()`)
4. Read the data from the file (`read()`)
5. Close the file (`close()`)

Stateful File Server

- Server may keep information on a table about previous requests of each client (e.g.):
  - file name (or file descriptor)
  - client permissions
  - current offset
  - id of previous request
- Server may support two additional operations:
  - open file, which returns a **file handle**
  - close file
- Read/write requests need to include only:
  - operation
  - client id (possibly)
  - file handle
  - number of bytes to transfer
  - data (only in write requests)

Upon a read request the server must:

1. Look up the file handle on the table, to get the file descriptor
2. Read the data from the file (`read()`)

Stateful Servers and Failures

- 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
    - E.g. if a client crashes before invoking `close()`
  - wrong interpretation of requests sent by other clients after the crash
    - If client id is reused (e.g. IP address and port number)

Stateful Servers and 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 a finite time interval: upon its expiration, the resource may be taken away, unless the client **renews** the lease

Stateless Servers and Message Loss

- Stateless servers are not immune to problems arising from failures:
  - message duplication may lead to handling the same request several times
    - operations must be **idempotent**, if the transport protocol does not ensure non-duplication of packets;
    - even if the transport protocol ensures non-duplication of packets, we may still need idempotent operations

      What if the connection breaks?
- How can stateful servers handle duplicated requests?
  - Need to be careful about client identification

Stateful Servers and Client Identification

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

Servers, State and Protocols

- **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;
  - Likewise, 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
    - Version 1.0 even used one TCP connection per request
  - Cookies are a device that allows a server to keep state about a client session (actually there are other types of cookies that may lead to abuse):
    - servers generate and send *cookies* to the clients
    - clients store the *cookies* received from serves
    - clients piggyback the *cookies* on HTTP requests

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 or even a slow host

Solution: highly application dependent, but we'll study some general techniques

Distribution is harder than concurrency

In concurrent (local) systems the programmer needs to consider all possible execution interleavings

In distributed systems the programmer needs **also** to consider all possible failures

- Distributed systems are inherently concurrent

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. check whether the 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