Distributed Systems Documentation

Release alpha

George K. Thiruvathukal and Joe Kaylor

June 29, 2015

You have reached the home of Distributed Systems (COMP 339-439) at Loyola University Chicago in the Computer Science Department. This course is normally taught by George K. Thiruvathukal.

Warning: These notes are still being written, so expect a few rough edges. But we're getting closer!

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Introduction and Issues

1.1.1 What is a distributed system?

"A collection of autonomous computers linked by a network with software designed to produce an integrated facility"

"A collection of independent computers that appear to the users of the system as a single computer"

1.1.2 Examples

Distributed systems

- Department computing cluster
- Corporate systems
- Cloud systems (e.g. Google, Microsoft, etc.)

Application examples

- Email
- News
- Multimedia information systems - video conferencing
- Airline reservation system
- Banking system
- File downloads (BitTorrent)
- Messaging

1.1.3 Illustration

1.1.4 Advantages of Distributed Systems vs. Centralized

Economics

- Commodity microprocessors have better price/performance than mainframes

Speed

- Collective power of large number of systems is potentially infinite

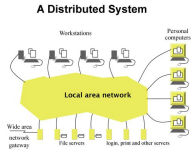


Fig. 1.1: A Distributed System

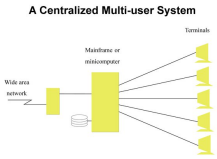


Fig. 1.2: A Centralized Multi-User System

Geographic and Responsibility Distribution

- Can provide better speed in geographic regions by not going over slower transoceanic cables

Reliability

- One machine's failure need not bring down the system

Extensibility

- Computers and software can be added incrementally

1.1.5 Advantages of Distributed Systems vs. Standalones

Data Sharing

- Multiple users can access common database, data files

Device/Resource Sharing

- e.g., printers, servers, other CPUs...

Communication

- Communication with other systems...

Flexibility

- Spread workload to different & most appropriate systems

Extensibility

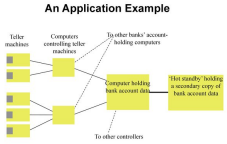


Fig. 1.3: An Application Example

- Add resources and software as needed

1.1.6 Disadvantages of Distributed Systems

Software

- Little software exists compared to PCs (for example) but the situation is improving with the cloud.

Networking

- Still slow and can cause other problems (e.g., when disconnected)

Security

- Data may be accessed by unauthorized users through network interfaces

Privacy

- Data may be accessed securely but without the owner's consent (significant issue in modern systems)

1.1.7 Key Characteristics

- Support for resource sharing
- Openness
- Concurrency
- Scalability
- Fault Tolerance (Reliability)
- Transparency

1.1.8 Resource Sharing

Share hardware, software, data and information

Hardware Devices

- printers, disks, memory, sensors

Software Sharing

- compilers, libraries, toolkits, computational kernels

Data

- databases, files

1.1.9 Resources Must be Managed

1.1.10 Openness

Determines whether the system can be extended in various ways without disrupting existing system and services

Hardware extensions (adding peripherals, memory, communication interfaces..)

Software extensions

- Operating System features

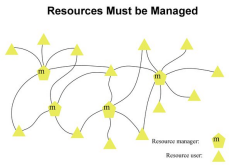


Fig. 1.4: Resources Must Be Managed

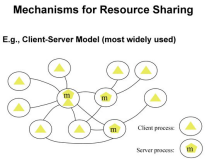


Fig. 1.5: Client-Server Model for Resource Sharing

- Communication protocols

Mainly achieved using published interfaces, standardization

- Great example of a distributed, standards-focused effort, http://www.ietf.org/

1.1.11 Open Distributed Systems

Are characterized by the fact that their key interfaces are published

- e.g., HTTP Protocol, https://www.ietf.org/rfc/rfc2616.txt

Based on the provision of a uniform interprocess communication mechanism and published interfaces for access to shared resources

Can be constructed from heterogeneous hardware and software.

1.1.12 Concurrency

- In a single system several processes are interleaved
- In distributed systems - there are many systems with one or more processors
- Many users simultaneously invoke commands or applications (e.g., Netscape..)
- Many server processes run concurrently, each responding to different client request, e.g., File Server

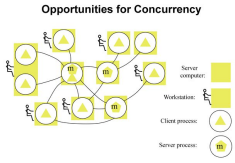


Fig. 1.6: Opportunities for Concurrency

1.1.13 Scalability

Scale of system

- Few PCs servers –> Dept level systems –> Local area network –> Internetworked systems —>Wide area network…
- Ideally - system and applications software should not (need to) change as systems scales

Scalability depends on all aspects

- Hardware
- Software
- Networks
- Storage

1.1.14 Fault Tolerance

- Ability to operate under failure(s) - possibly at a degraded performance level
- Two Approaches - Hardware redundancy - use of redundant components - Software Recovery - design of programs to recover
- In distributed systems - servers can be replicated - databases may be replicated - software recovery involves the design so that state of permanent data can be recovered
- Distributed systems, in general, provide a high(er) degree of availability

1.1.15 Transparency

Transparency "is the concealment from the user of the separation of components of a distributed system so that the system is perceived as a whole".

Examples

- Access Transparency - enables local and remote objects to be accessed using identical operations (e g., read file..)
- Location transparency - location of resources is hidden
- Migration transparency - resources can move without changing names
- Replication Transparency - users cannot tell how many copies exist
- Concurrency Transparency - multiple users can hare resources automatically
- Parallelism Transparency - activities can happen in parallel without user knowing about it
- Failure Transparency - concealment of faults

1.1.16 Design Issues

- Openness
- Resource Sharing
- Concurrency
- Scalability

- Fault-Tolerance
- Transparency
- High-Performance

1.1.17 Issues arising from Distributed Systems

- Naming - How to uniquely identify resources
- Communication - How to exchange data and information reliably with good performance
- Software Structure - How to make software open, extensible, scalable, with high-performance
- Workload Allocation - Where to perform computations and various services
- Consistency Maintenance - How to keep consistency at a reasonable cost

1.1.18 Naming

- A resource must have a name (or identifier) for access
- Name: Can be interpreted by user, e.g., a file name
- Identifier - Interpreted by programs, e.g., port number

1.1.19 Naming - Name Resolution

- "resolved" when it is translated into a form to be used to invoke an action on the resource
- Usually a communication identified PLUS other attributes
- E.g., Internet communication id
  - host id:port no
  - also known as "IP address:port no"
  - 192:130.228.6:8000
- Name resolution may involve several translation steps

1.1.20 Naming - Design Considerations

- Name space for each type of resource
  - e.g., files, ports, printers, etc.
- Must be resolvable to communication Ids
  - typically achieved by names and their translation in a "name service"
  - You must have come across "DNS" when using the WWW!!
- Frequently accessed resources, e.g., files are resolved by resource manager for efficiency
- Hierarchical Name Space - each part is resolved relative to current context, e.g., file names in UNIX

1.1.21 Communication

Communication is an essential part of distributed systems - e.g., clients and servers must communicate for request and response

Communication normally involved - transfer of data from sender to receiver - synchronization among processes

Communication accomplished by message passing

Synchronous or blocking - sender waits for receiver to execute a receive operation

Asynchronous or non-blocking

1.1.22 Types of Communication

- Client-Server
- Group Multicast
- Function Shipping
- Performance of distributed systems depends critically on communication performance
- We will study the software components involved in communication

1.1.23 Client-Server Communication

- Client sends request to server process
- Server executes the request
- Server transmits a reply and data, e.g., file servers, web server...

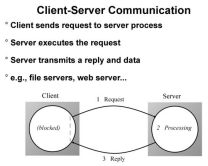


Fig. 1.7: Client-Server Communication

1.1.24 Client-Server Communication

- Message Passing Operations
  - send
  - receive
- Remote Procedure Call (RPC)
  - hides communication behind procedure call abstraction
  - e.g., read(fp,buffer,....)
  - Files reside with the server, thus there will be communication between client and server to satisfy this request

1.1.25 Group Multicast

- A very important primitive for distributed systems
- Target of a message is a group of processes
  - e.g., chat room, I sending a message to class list, video conference
- Where is multicast useful?
  - Locating objects - client multicasts a message to many servers; server that can satisfy request responds
  - Fault-tolerance - more than one server does a job; even if one fails, results still available
  - Multiple updates
- Hardware support may or may not be available
  - if no hardware support, each recipient is sent a message

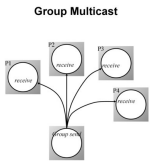


Fig. 1.8: Group Multicast

1.1.26 Software Structure

- In a centralized system, O/S manages resources and provides essential services
- Basic resource management
  - memory allocation and protection
  - process creation and processor scheduling
  - peripheral device handling
- User and application services
  - user authentication and access control (e.g., login)
  - file management and access facilities
  - clock facilities

1.1.27 Distributed System Software Structure

- It must be easy to add new services (flexibility, extensibility, openness requirements)
- Kernel is normally restricted to
  - memory allocation
  - process creation and scheduling
  - interposes communication
  - peripheral device handling

- E.g., Microkernels - represent light weight O/S, most services provided as applications on top of microkernels

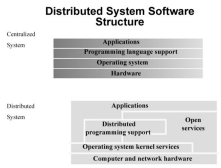


Fig. 1.9: Distributed System Software Structure

1.1.28 Consistency Management

- When do consistency problems arise?
  - concurrency
  - sharing data
  - caching
- Why cache data?
  - for performance, scalability
- How?
  - Subsequent requests (many of them) need not go over the NETWORK to SERVERS
  - better utilized servers, network and better response
- Caching is normally transparent, but creates consistency problems

1.1.29 Caching

- Suppose your program (pseudocode) adds numbers stored in a file as follows (assume each number is 4 bytes:

```
for I= 1, 1000
       tmp = read next number from file
       sum = sum + tmp
end for
```

- With no caching, each read will go over the network, which will send a new 4 byte number. Assuming 1 millisecond (ms) to get a number, requres a total of 1s to get all of the numbers.
- With caching, assuming 1000 byte pages, 249 of the 250 reads will be local requests (from the cache).

1.1.30 Consistency

- Update consistency
  - when multiple processes access and update data concurrently
  - effect should be such that all processes sharing data see the same values (consistent image)
  - E.g., sharing data in a database
- Replication consistency
  - when data replicated and once process updates it

  - All other processes should see the updated data immediately
  - e.g., replicated files, electronic bulletin board
- Cache consistency
  - When data (normally at different levels of granularity, such as pages, disk blocks, files. . .) is cached and updates by one process, it must be invalidated or updated by others
  - When and how depends on the consistency models used

1.1.31 Workload Allocation

- In distributed systems many resources (e.g., other workstations, servers etc.) may be available for "computing"
- Capacity and size of memory of a workstation or server may determine what applications may are able to run
- Parts of applications may be run on different workstations for parallelism (e.g., compiling different files of the same program)
- Some workstations or servers may have special hardware to do certain types of applications fast (e.g., video compression)
- Idle workstations may be utilized for better performance and utilization

1.1.32 Processor Pool Model

In a processor pool model, processes are allocated to processors for their lifetime (e.g the Amoeba research O/S supports this concept).

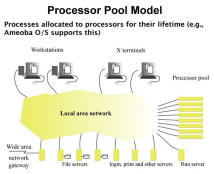


Fig. 1.10: Processor Pool Model

1.1.33 Quality-of-Service

Quality of Service (a.k.a. QoS) refers to performance and other service expectations of a client or an application.

- Performance
- Reliability and availability
- security

Examples where this is important.

- Voice over IP (VOIP) and telephony
- Video (e.g. Netflix and friends)

1.2 Networking Primer

- Introduction to networks
- Protocols and Layers
- Interprocess Communication
  - Sockets
  - RPC
- Network Case Studies (Ethernet, ATM)
- Protocol Case Studies (TCP/IP, Client-Server)

Additional reading:

- See http://intronetworks.cs.luc.edu/html by our colleague, Dr. Peter Dordal.

1.2.1 Networks

- Communication between semi-autonomous computers
- Attached to host system by an adapter

FIGURE

- Communication networks provide an infrastructure for communication in distributed systems
- Infrastructure is required at various levels
  - Cables/wires
  - Switches
  - Interfaces
  - Software components at various levels: protocol managers, network managers
- The entire collection of the above is a "communication system"

1.2.2 Many Types of Networks

Physical Media

- copper wires (Ethernet, RS232-C, V.32, etc.)
- fiber optics (ATM, FDDI)
- air (IR, Radio, micro-wave)
- Speeds (link not aggregate)
- **low**
  - modems (few k bits/sec)
  - pagers
- **medium**
  - Ethernet (10-100 Mbps)
  - Token Ring (10 Mbps)

- **high**
  - ATM (155-655 Mbps)
  - Myrinet (600 Mbps)
  - SONET (OC-48 - 2488 Mbps)

1.2.3 Many Types of Networks

- **Local Area Networks**
  - Relatively high-speed (oh yeah!)
  - Normally single building, campus, Office
  - Most of the time direct (does not mean all-to-all) connection between computers
  - Low Latency
  - Eg., Ethernet, FDDI, IBM Token Ring
- **Wide Area Networks**
  - Msgs at lower speeds between systems separated by large distances
  - Communication circuits connected by "Packet switching" computers, that also manage the network
  - Messages are routed by "packet switches"
  - E.g., ISDN, BISDN, ATM

1.2.4 Many Types of Networks

- **Metropolitan Area Networks (MANs)**
  - within cities, towns, use fiber-optics cables
  - recent, to carry voice, video...

1.2.5 Internetworks

- Remember - distributed systems require extensibility
- Must be able to connect/link networks together => Internetworks
- Normally achieved by linking component networks with dedicated routers OR
- by connecting them by general purpose computers called "gateways"
- protocols that support addressing and transmission between these networks are added
- Internet is a GIANT Wide-Area-Network connecting thousands of component networks.

1.2.6 Performance Issues

- **Performance parameters**
  - Latency - time needed to transfer an empty message between two systems - normally measures software delay and transit time

  - **What is software delay?**
    - time to access the network, I.e., put bits on the network from the time a message is sent by the application and time to retrieve the bits and supply the msg to the receiver
    - software delay can be quite large because message has to go through several layers (later)
  - **Bandwidth (or Data Transfer Rate) seen by a message**
    - rate at which data is transferred over the network once the transmission started
  - Network Bandwidth - Volume of traffic per unit time transferred across the network
  - Quality of Service (QoS) - other guarantees such as delay and b/w guarantees, reliability and availability guarantees
  - Assuming no delays due to congestion

1.2.7 Example- A Typical Campus Network

FIGURE

1.2.8 Network Topologies

- How are the communicating objects connected
- Fully connected - link between all sites
- **Partially connected**
  - links between subset of sites
  - can be an arbitrary graph
- **Hierarchical networks**
  - network topology looks like a tree
  - internal nodes route messages between different sub-trees
  - if an internal node fails, children can not communicate with each other
  - star network - hierarchical network with single internal node

1.2.9 Network Topologies

FIGURE

1.2.10 A Network is not an Island

- Reason for networks is to share information
  - must be able to communicate in a common language
  - **called protocols**
    - The nice thing about protocols is that there are so many of them!
- Protocols
  - **must be unambiguous and followed exactly**

        - **rule of thumb for good protocol implementations**
            - be rigorous is what you generate
            - be liberal in what you accept
    - **there are many different aspects to protocols**
        - electrical through web services

1.2.11 Design Issues In Layers

- **Rules for data transmission (Protocol)**
    - full Vs. half duplex
    - error control (detection, correction, etc.)
    - flow control (rate matching, overuse of shared resources)
    - message order (do things arrive in the same order as sent?)
- **Abstractions for communications**
    - **end points for communication**
        - switches, nodes, processes, threads in a process
        - how are these end points named (addresses)?
    - service providers and service users
- **Service Primitives**
    - operations performed by a layer
    - events and their actions
    - request, indication, response, confirm

1.2.12 Protocols are divided into layers

- **ISO - seven layer reference model**
    - Application
    - Presentation
    - Session
    - Transport
    - Network
    - Link
    - Physical
- **TCP/IP - four layer model**
    - application
    - transport
    - network (internet)
    - link

1.2.13 Physical Layer

- Goal: Raw bits over a communication channel
- **Sample Issues:**
  - how to encode a 0 Vs. 1?
  - what voltage should be used?
  - how long does a bit need to be signaled?
  - what does the cable, plug, antenna, etc. look like?
- **Examples:**
  - modems
  - "knock once for yes, twice for no"
  - X.21

1.2.14 Data Link Layer

- Goal: transmit error free frames over the physical link
- **Sample Issues:**
  - how big is a frame?
  - can I detect an error in sending the frame?
  - what demarks the end of the frame?
  - how to control access to a shared channel?
- **Examples:**
  - Ethernet framing
  - CSMA/CD

1.2.15 The Network Layer

- Goal: controlling operations of the subset
- **Sample Issues:**
  - how route packets that have to travel several hops?
  - control congestion - too many messages at once
  - accounting - charge for use of the network
  - fragment or combine packets depending on rules of link layer
- **Examples:**
  - IP
  - X25

1.2.16 The Transport Layer

- **Goal: accurately transport session data in order**
  - end points are the sending and receiving machines
- **Sample Issues:**
  - how to order messages and detect duplicates
  - error detection (corrupt packets) and retransmission
  - connectionless or connection-oriented
- **Examples:**
  - TCP (connection-oriented)
  - UDP

1.2.17 The Session & Presentation Layers

- Goal: common services shared by several applications
- **Sample Issues:**
  - network representation of bytes, ints, floats, etc.
  - encryption?? (this point is subject to lots of debate)
  - synchronization
- **Examples:**
  - eXternal Data Representation (XDR)

1.2.18 Application Layer

- Goal: common types of exchanges standardized
- **Sample Issues:**
  - when sending email, what demarks the subject field
  - how to represent cursor movement in a terminal
- **Examples:**
  - Simple Mail Transport Protocol (SMTP)
  - File Transfer Protocol (FTP)
  - Hyper-Text Transport Protocol (HTTP)
  - Simple Network Management Protocol (SNMP)
  - Network File System (NFS)
  - Network Time Protocol (NTP)
  - Net News Transport Protocol (NNTP)
  - X (X Window Protocol)

1.2.19 Interprocess Communication:

- **Sockets & RPC (Basic operations)**
  - Send
  - Receiver
  - Synchronize
  - => Send must specify destination
  - => Clients need to know an identifier for communicating with another process (e.g., server)

1.2.20 Reliability

- "Unreliable Message" - single msg sent from sender to recipient without acknowledgment (e.g., UDP)
- Processes that use unreliable messages are responsible for enforcing correct/reliable message passing
- **Reliability introduces overhead**
  - need to store state information at the source and destination
  - transmit extra messages (e.g., ack)
  - latency (for processing information related to reliability)

1.2.21 Mapping Data to Messages

- Programs have data structures
- Messages are self-contained sequence of bytes
- **=> For communication**
  - data structures must be flattened before sending
  - rebuilt upon receipt
- Problem: How does the receiver know how the sender has flattened?
- What if sender and receiver have different representations?
- => Follow standard (possibly external) data format - or the one which has been agreed upon between sender and receiver in advance

1.2.22 Marshaling

- Process of taking a collection of data items and assembling them into a form for transmission
- Unmarshaling - Disassemble message upon receipt
- Normally programs supplied with standards
- For example msg - 5 smith 6 London 1934
- In C, `sprintf()` (data item -> array of characters), `sscanf()` for opposite:

The following shows how to marshall some data using `sprintf()`:

```
char *name = "smith", place = "London"; int year = 1934
sprintf(message, "%d %s %d %s %d", strlen(name), name, strlen(place), place, years);
```

Can you think of how to write the unmarshalling version using `sscanf()`?

1.2.23 Case Study: UNIX Interprocess Communication (IPC)

- IP C provided as systems calls implemented over TCP and UDP
- Message destinations - Socket addresses (Internet address and port id)
- Communication operations based on socket pairs (sender and receiver)
- Msgs queued at sender socket until network protocol transmits them and ack
- Before communication can occur - recipient must BIND its socket descriptor to a socket address

1.2.24 Example - Simple TCP Messaging Framework (from HPJPC)

- TCP/IP example
- simple messaging service where the client/server exchange Message objects containing key/value parameters
- can send all primitive types or binary-encoded data
- Key classes
  - Message
  - MessageClient
  - MessageServer and MessageServerDispatcher (handles concurrent requests)
  - MessageService interface (for building your own services)
- Example Service
  - DateService
  - DateClient

1.2.25 Message class

1.2.26 MessageClient class

1.2.27 MessageServer class

1.2.28 MessageServerDispatcher class

1.2.29 DateService using Message Classes/Interfaces

1.2.30 DateClient using Message Classes/Interfaces

1.2.31 Sockets Communication Using Datagram

- "socket" call to create and a get a descriptor
- Bind call to bind socket to socket address (internet address & port number)

- Send and receive calls use socket descriptor to send receive messages
- UDP, no ack

FIGURE

1.2.32 Stream Communication

FIGURE

- First need to establish a connection between sockets
- Asymmetric because one would be listening for request for connection and the other would be asking
- Once connection, data communication in both directions

1.2.33 Remote Procedure Call

- 17. How do me make "distributed computing look like traditional (centralized) computing"?
- **Simple idea - Can we use procedure calls? Normally,**
  - A calls B –> A suspended, B executes –> B returns, A executes
  - Information from A (caller) to B (callee) transferred using parameters
  - Somewhat easier since both caller and callee execute in the same address space
- **But in Distributed systems - the callee may be on a different system**
  - ==> Remote Procedure Call (RPC)
  - NO EXPLICIT MESSAGE PASSING (which is visible to the programmer)

1.2.34 Remote Procedure Call (RPC)

- Although no message passing (at user level) - parameters must still be passed - results must still be returned!
- ==> Many issues to be addressed - Look at an example to understand some issues

```
count = read(fd, buf, nbytes)
[fd-file pointer (int), buf-array of chars, nbytes-integer)
```

FIGURE

1.2.35 Observations

- parameters (in C): call-by-reference OR call-by-value
- Value parameter (e.g., fd, nbytes) copied onto stack (original value not affected)
- Value parameter is just an initialized variable on stack for callee
- **Reference parameter (array buf) is not copied –> pointer to it is passed (buf's address)**
  - Original values modified
- Many options are language dependent but we will ignore them. . .
- How to deal with these situations?

1.2.36 RPC

- **Goal: Make RPC look (as much as possible) like local procedure call, that is,**
  - call should not be aware of the fact that the callee is on a different machine (or vice versa)
- **Look at the read call again and various involved components**
  - read routine is extracted from the library by linker and inserted into application object code
  - call read —Parameter onto stack–> kernel trap –> operation –POP–> return
  - programmer does not know all this
- in RPC —> read is remote ==> no way to put parameters on stack (no shared space/memory!)
- Solution: In the library keep "client stub" which acts like "read"
- So how does it work?

1.2.37 RPC Mechanisms

- Client-stub packs parameters
- Ships them to "server-

1.2.38 RPC Steps

1. client calls client stub in normal fashion
2. client stub builds msg and traps to kernel
3. kernel sends msg to remote kernel
4. remote kernel gives msg to server stub
5. server stub unpacks parameters and calls server
6. server processes and returns results to stub
7. server stub packs result in msg and traps to kernel
8. remote kernel sends msg to client kernel
9. client kernel gives msg to client stub
10. stub unpacks results and returns to client

1.2.39 Design Issues

- Parameter passing
- Binding
- **Reliability/How to handle failures**
  - messages losses
  - client crash
  - server crash
- Performance and implementation issues

- Exception handling
- Interface definition

1.2.40 Parameter Passing

- Some issues similar to messages passing
- Example below- what if clients and servers have different representations (Little endian vs big endian)

1.2.41 Parameter Passing

- **How to solve the problem?**
  - client and server know parameter type
  - **msg will have n+1 fields**
    - 1 - procedure identifier
    - n - procedure parameters

1.2.42 Binding

- 17. **How does a client locate the server?**
  - **Hardwire?**
    - inflexible
    - need to recompile all codes affected for any change
  - **Dynamic Binding**
    - formal specification of server

1.2.43 Use of Specification

- **Input to the stub generator - produces both client and server stub**
  - client stub linked to client function
  - server stub linked to server function
- **Server exports the server interface (initialize())**
  - server sends msg to binder to know it is up (registration)
  - **server gives the binder**
    - name
    - version number
    - unique id
    - handle (system dependent - IP address, Ethernet address..)

1.2.44 Locating the Server

- First call to RPC of function
- Client stub sees not bound to server
- Client stub sends msg to binder to "import" interface
- If server exists, binder gives unique id and handle to client stub
- Client stub uses these for communication
- **Method flexible**
  - can handle multiple servers with same interface
  - binder can poll servers to see if up or deregister them if down for fault tolerance
  - can enforce authentication
- **Disadvantage**
  - overhead of interface export/import
  - binder may be a bottleneck in large systems

1.2.45 How to Handle Failures

- **Types of possible failures in RPC systems**
  1. client unable to locate server
  2. request message from client to server is lost
  3. reply message from server to client is lost
  4. server crashes after receiving a request
  5. client crashes after sending a request (^c!!)
- 17. What are the semantics?
- 17. How close can we get to the goal of transparency?

1.2.46 Client Cannot Locate Server

- **Why?**
  - server may be down
  - new version of server (using new stubs..) but older client ==> binder cannot match
- **Solutions**
  - **respond with error type "cannot locate server"**
    - · simple
    - not general (what if the error code, e.g. -1, is also a result of computation?)
  - **raise exception**
    - some languages allow calling special procedures for error
    - not all languages support this

        * destroys transparency

1.2.47 Lost Request Message

- **Time Out**
    - Kernel starts timer when request sent
    - If timer expires, resend message
    - If message was lost - server cannot tell the difference
    - If message lost too many times ==> "cannot locate server"

1.2.48 Lost Reply Message

- More difficult to handle
- Rely on timer again?
- Problem: Client's kernel doesn't know why no answer!
- **Must distinguish between**
    - request/reply got lost?
    - server slow
- **Why?**
    - some operations may be repeated without problems (e.g., reading a block from the same position in file–no side effects)
    - property - "idempotent"

1.2.49 Lost Reply Message

- **What if request is not idempotent?**
    - e.g., transferring 500 thousand dollars from your account
    - do it five times and you are broke!
- Solution - Client kernel uses a sequence number (needs to maintain state) for each request
- Have a bit in message to distinguish initial vs. retransmissions

1.2.50 Server Crashes

- **Depends on when server crashes**
    - After execution
    - After receiving message but BEFORE execution
- Solutions differ

FIGURE

1.2.51 Server Crashes

- But the client cannot tell the difference!
- **Solutions?**
  - **Wait until server reboots (or rebind)**
    - try operation again and keep trying until success
    - "at least once semantics"
  - **Give up immediately and report failure**
    - "at most once semantics"
  - **Guarantee nothing**
    - (-) RPC may be tried from 0 - any no
    - (+) easy to implement
  - But none of the above attractive
  - **What we want is "exactly once semantics"**
    - no way to insure this

1.2.52 Client Crashes

- **Client sends a request and crashes**
  - computation active - but no parent active
  - unwanted computation called "orphan"
- **Orphan's can create problems**
  - wasted resources
  - locked files?
  - client reboots - does RPC - reply from orphan comes =>confusion!
- **Solutions (Extermination)**
  - client stub logs (on disk) request before sending
  - after reboot check log - kill any orphan
  - (+) simple
  - (-) too expensive (each RPC requires disk access!)
- what if orphans do RPC => grand orphans => difficult to kill all

1.2.53 Client Crashes

- **Reincarnation**
  - divide time into numbered slots (epoch)
  - when client reboots, it broadcasts to all machines with new slot
  - all remote computations killed

    - if network partitioned, some orphans will remain - but will be detected later
- **Gentle Reincarnation**
    - locate the owner of the orphan first
    - if not found, kill computations

1.2.54 Acknowledgments

- How to acknowledge when RPC packets are broken up?

FIGURE

1.2.55 Flow Control

- Network Interface Chips (NICs) can send message fast
- But receiving more difficult due to finite buffer
- **Overrun can occur when**
    - NIC serving one packet
    - another arrives
- No overrun possible in stop-and-wait (assuming single sender)
- **Sender can insert gaps (assume n buffer capacity)**
    - send n packets
    - gap
    - send n packets
- Performance
- Critical Path

1.2.56 Performance

FIGURE(s) that need updating.

Where is the time spent? Example firefly workstation

1.3 Concurrency and Threads

1.3.1 Threads vs Processes

- **Traditional Operating Systems have Processes**
    - Each process has its own address space
    - Single thread of control
    - When a process makes a system call, the process blocks until the call completes.
- **In many situations it is desirable to have multiple threads of control for:**

  - Explicit parallelism - to take advantage of multiple processors
  - **To allow computation to continue while waiting for system calls to return**
    * example: one thread reads a file into a buffer, another thread compresses the buffer, and a final thread writes out a compressed file.
  - To reduce the cost of context switching
  - Implicit parallelism - to keep a single processor busy

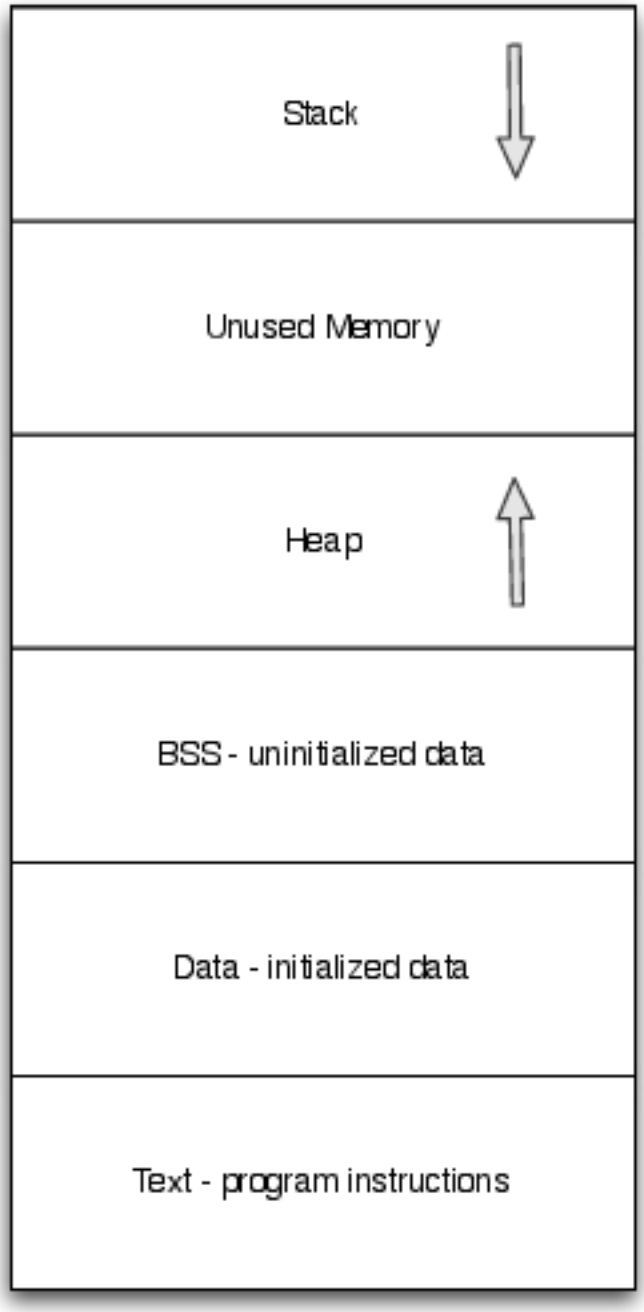
1.3.2 What are Processes, Threads?

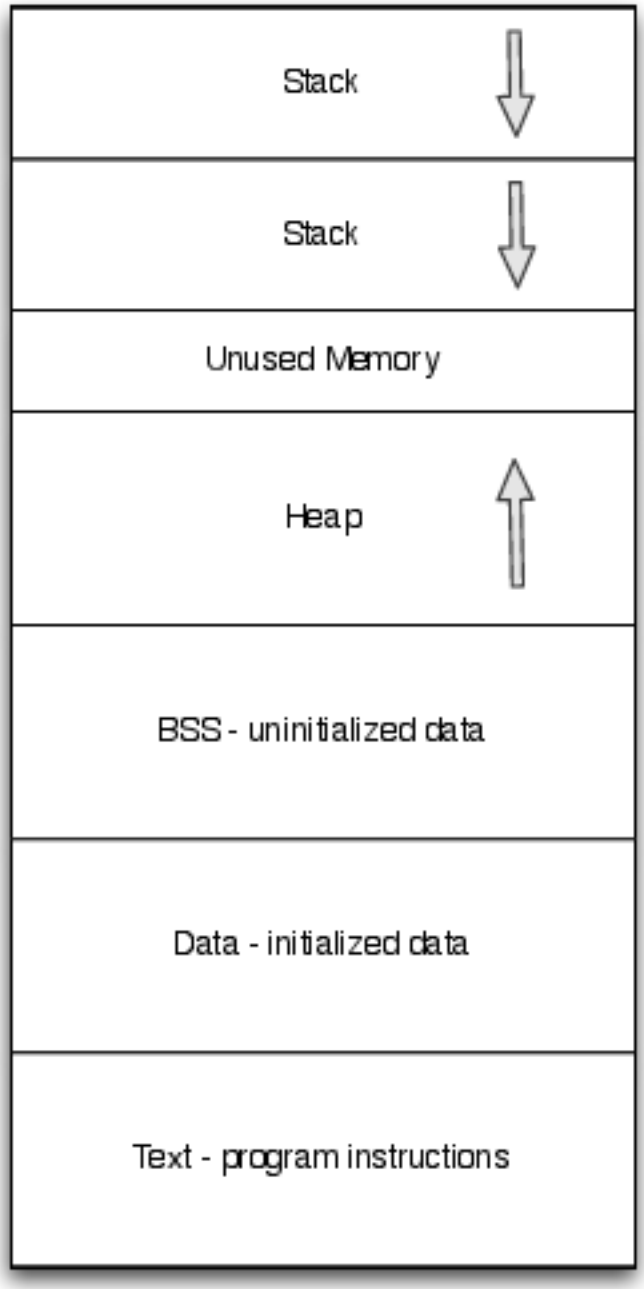
- **Processes have:**
  - Program counter
  - Stack
  - Register set
  - An exclusive virtual address space
  - Sandboxing from other process except to the extent that the process participates in IPC
- **Threads:**
  - Threads can be thought of (and are often referred to as) lightweight processes
  - Provide multiple threads of control
  - Have multiple program counters
  - Have multiple stacks
  - One parent process, the virtual address space is shared across processes
  - Each thread runs sequentially
  - In a given process with N threads, 0-i threads may be blocked, and 0-k threads are runnable or running.

1.3.3 Common Threading Use Cases

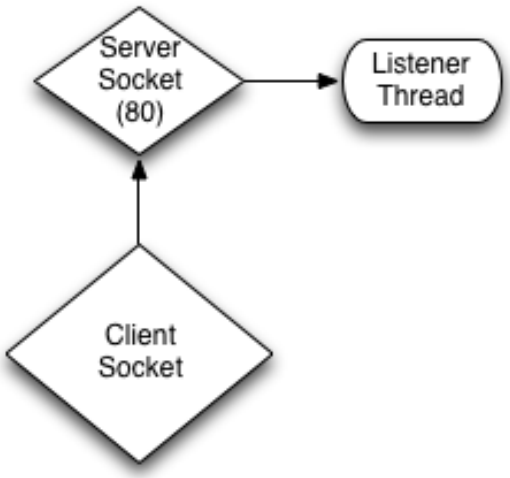
- **Client / Server - ex: file servers:**
  - One thread listens on a network socket for clients
  - When a client connects, a new thread is spawned to handle the request. This permits several clients to connect to the file server at one time because each request is handled by a separate thread of execution.
  - To send the file data, two threads can be used the first can read from the file on disk and the second can write the read buffers to the socket.
- Example Client-Server - a TCP echo server

```
1       public class TcpServer
2       {
3               private Socket _socket;
4               private Thread _serverThread;
5
6               public TcpServer() {
7                       _socket = new Socket (AddressFamily.InterNetwork, SocketType.Stream, Protocol
8                       _serverThread = new Thread (Server);
9                       _serverThread.Start ();
```

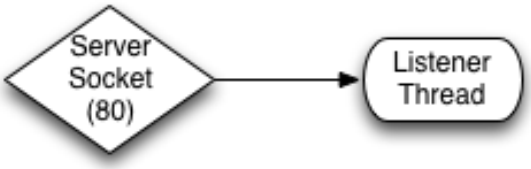

Stack
Unused Memory
Heap
BSS - uninitialized data
Data - initialized data
Text - program instructions

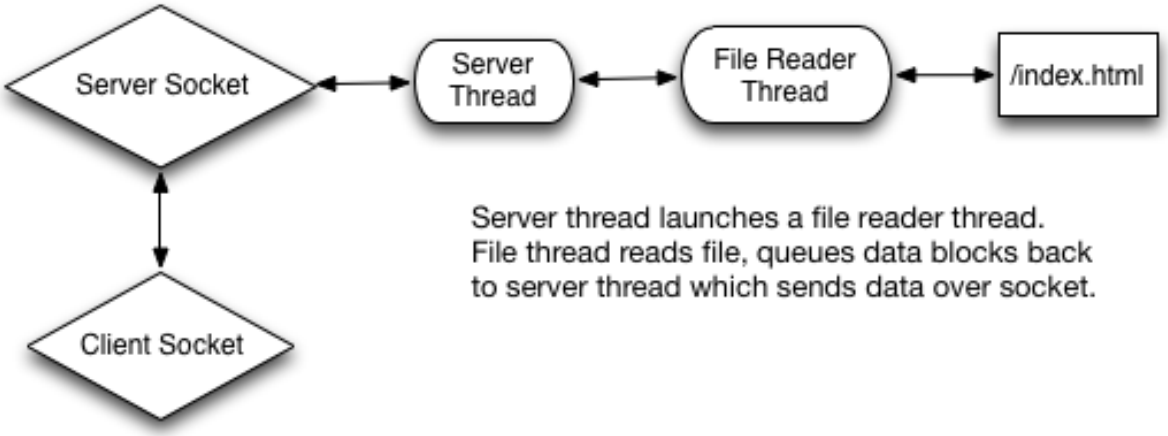

Stack
Stack
Unused Memory
Heap
BSS - uninitialized data
Data - initialized data
Text - program instructions

Server Socket receives packet, listener thread handles it.



Listener thread launches server thread and hands it
the new socket that is connected with the client





Server thread launches a file reader thread.
File thread reads file, queues data blocks back
to server thread which sends data over socket.

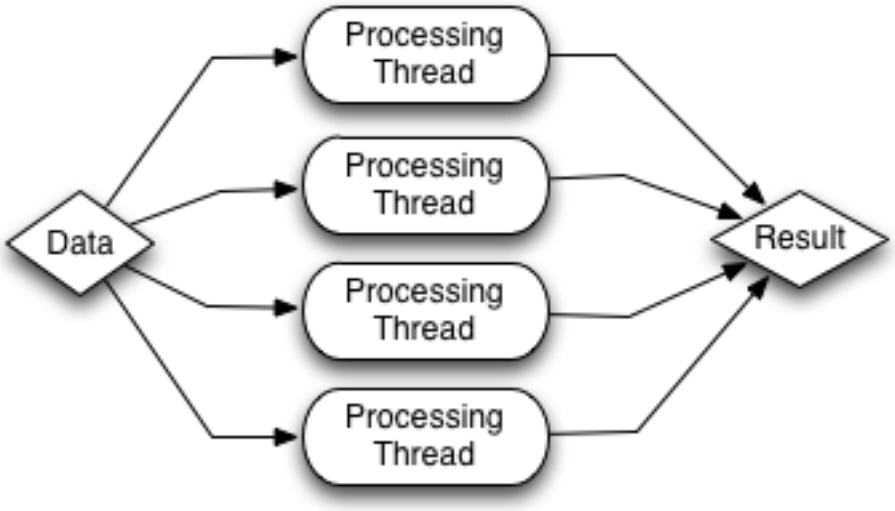
```
                }

                private void Server() {
                        _socket.Bind (new IPEndPoint (IPAddress.Any, 8080));
                        _socket.Listen (100);
                        while (true) {
                                var serverClientSocket = _socket.Accept ();
                                new Thread (Server) { IsBackground = true }.Start (serverClientSocket
                        }
                }

                private void ServerThread(Object arg) {
                        try {
                                var socket = (Socket)arg;
                                socket.Send (Encoding.UTF8.GetBytes ("Echo"));
                        } catch(SocketException se) {
                                Console.WriteLine (se.Message);
                        }
                }
        }

        public class TcpClient {
                public void ConnectToServer() {
                        var socket = new Socket (AddressFamily.InterNetwork, SocketType.Stream, Proto
                        socket.Connect (new IPEndPoint (IPAddress.Parse("127.0.0.1"), 8080));
                        var buffer = new byte[1024];
                        var receivedBytes = socket.Receive (buffer);
                        if (receivedBytes > 0) {
                                Array.Resize (ref buffer, receivedBytes);
                                Console.WriteLine (System.Text.Encoding.UTF8.GetString (buffer));
                        }
                }
        }
```

- **Parallel computation:**
  - An algorithm is designed to solve some small part or subproblem of a larger problem
  - To the extent that the subproblems are not inter-dependent, they can be executed in parallel
  - Multiple threads can work against a common task queue.

Warning: The focus of this course is on *distributed* (not *parallel*) systems. Nevertheless, you may find that you want to take advantage of parallel computing in your work. We encourage you to read Christopher and Thiruvathukal, http://hpjpc.googlecode.com, which contains many examples of parallel algorithms in Java. You may also find Läufer, Lewis, and Thiruvathukal's Scala workshop tutorial helpful. See http://scalaworkshop.cs.luc.edu.

- Example Parallel Computation - factoring an integer

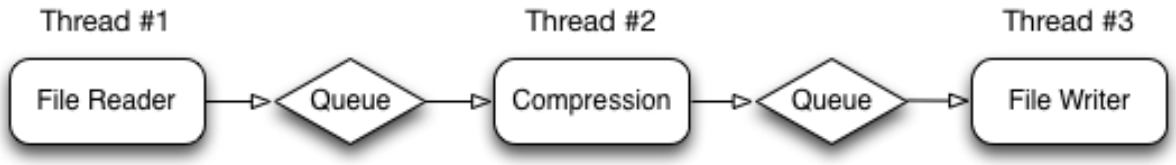
```
        public class ParallelComputationWorkItem {
                public readonly long LowerBound;
                public readonly long UpperBound;
                public readonly long Number;
                public readonly ICollection<long> Divisors;
                public ParallelComputationWorkItem(long lower, long upper, long number, ICollection<l
                        LowerBound = lower;
                        UpperBound = upper;
                        Number = number;
                        Divisors = divisors;
```




```
                }
        }

        public class ParallelComputation {

                public void PrintFactors(long number) {
                        var threads = new List<Thread>();
                        var divisors = new List<long> ();
                        var cpuCount = Environment.ProcessorCount;
                        long lower = 1;
                        for (var i = 0; i < cpuCount; i++) {
                                var thread = new Thread (Worker);
                                var upper = lower + (number / cpuCount);
                                thread.Start(new ParallelComputationWorkItem(lower, upper, number, di
                                threads.Add (thread);
                                lower = upper;
                        }
                        foreach (var thread in threads) {
                                thread.Join ();
                        }
                        Console.WriteLine ("Divisors - ");
                        foreach (var divisor in divisors) {
                                Console.WriteLine (divisor);
                        }
                }

                private void Worker(object args) {
                        var workItem = (ParallelComputationWorkItem)args;
                        for (var i = workItem.LowerBound; i < workItem.UpperBound; i++) {
                                if (workItem.Number % i == 0) {
                                        lock (workItem.Divisors) {
                                                workItem.Divisors.Add (i);
                                        }
                                }
                        }
                }
```

```
47          }
```

- **Pipeline processing:**
  - An algorithm must be executed in several stages that depend upon each other.
  - For example if there are three stages, then three threads can be launched for each of the stages. As the first thread completes some part of the total work, it can pass it to a queue for the second stage to be processed by the second thread. At this time, the first thread and second thread can work on their own stages in parallel. The same continues to the third thread for the third stage of computation.



- Example Pipeline Processing - file compression

```
1       public class PipelineComputation {
2               private readonly Queue<byte[]> _readData;
3               private readonly Queue<byte[]> _compressionData;
4               private volatile bool _reading = true;
5               private volatile bool _compressing = true;
6
7               public PipelineComputation () {
8                       _readData = new Queue<byte[]>();
9                       _compressionData = new Queue<byte[]>();
10              }
11
12              public void PerformCompression() {
13                      var readerThread = new Thread (FileReader);
14                      var compressThread = new Thread (Compression);
15                      var writerThread = new Thread (FileWriter);
16                      readerThread.Start ();
17                      compressThread.Start ();
18                      writerThread.Start ();
19                      readerThread.Join ();
20                      compressThread.Join ();
21                      writerThread.Join ();
22              }
23
24              private void FileReader() {
25                      using (var stream = new FileStream("file.txt", FileMode.Open, FileAccess.Read
26                              int len;
27                              var buffer = new byte[1024];
28                              while ((len = stream.Read(buffer, 0, buffer.Length)) > 0) {
29                                      if (len != buffer.Length) {
30                                              Array.Resize (ref buffer, len);
31                                      }
32                                      lock (_readData) {
33                                              while (_readData.Count > 10) {
34                                                      Monitor.Wait (_readData);
35                                              }
36                                              _readData.Enqueue(buffer);
37                                              Monitor.Pulse (_readData);
```

```
                                }
                        }
                }
                _reading = false;
        }

        private void Compression() {
                var workLeft = false;
                while (_reading || workLeft) {
                        workLeft = false;
                        byte[] dataToCompress = null;
                        lock (_readData) {
                                while (_reading && _readData.Count == 0) {
                                        Monitor.Wait (_readData, 100);
                                }
                                workLeft = _readData.Count > 1;
                                if (_readData.Count > 0) {
                                        dataToCompress = _readData.Dequeue ();
                                }
                        }
                        if (dataToCompress != null) {
                                var compressed = Compress(dataToCompress);
                                lock (_compressionData) {
                                        while (_compressionData.Count > 10) {
                                                Monitor.Wait (_compressionData, 100);
                                        }
                                        _compressionData.Enqueue (compressed);
                                        Monitor.Pulse (_compressionData);
                                }
                        }
                }
                _compressing = false;
        }

        private static byte[] Compress(byte[] data) {
                var memStream = new MemoryStream ();
                using(var compressionStream = new GZipStream(memStream, CompressionMode.Compr
                        compressionStream.Write(data, 0, data.Length);
                }
                return memStream.ToArray ();
        }

        private void FileWriter() {
                using (var stream = new FileStream("file.gz", FileMode.OpenOrCreate, FileAcce
                        var workLeft = false;
                        while (_compressing || workLeft) {
                                workLeft = false;
                                byte[] compressedData = null;
                                lock (_compressionData) {
                                        while (_compressionData.Count == 0 && _compressing)
                                                Monitor.Wait (_compressionData, 100);
                                        }
                                        workLeft = _compressionData.Count > 1;
                                        if (_compressionData.Count > 0) {
                                                compressedData = _compressionData.Dequeue ();
                                        }
                                }
                                if (compressedData != null) {
```

```
                                                stream.Write (compressedData, 0, compressedData.Lengt
                                        }
                                }
                        }
                }
        }
```

- Example Pipeline Processing - a more concise and language friendly file compression

```
        public class ConcisePipelineComputation
        {
                public ConcisePipelineComputation () {
                }

                public void PerformCompression() {
                        var fileBlocks = new ThreadedList<byte[]>(FileReader());
                        var compressedBlocks = new ThreadedList<byte[]> (Compression(fileBlocks));
                        FileWriter (compressedBlocks);
                }

                private IEnumerable<byte[]> FileReader() {
                        using (var stream = new FileStream("file.txt", FileMode.Open, FileAccess.Read
                                int len;
                                var buffer = new byte[1024];
                                while ((len = stream.Read(buffer, 0, buffer.Length)) > 0) {
                                        if (len != buffer.Length) {
                                                Array.Resize (ref buffer, len);
                                        }
                                        yield return buffer;
                                }
                        }
                }

                private IEnumerable<byte[]> Compression(IEnumerable<byte[]> readBuffer) {
                        foreach (var buffer in readBuffer) {
                                yield return Compress (buffer);
                        }
                }

                private static byte[] Compress(byte[] data) {
                        var memStream = new MemoryStream ();
                        using(var compressionStream = new GZipStream(memStream, CompressionMode.Compr
                                compressionStream.Write(data, 0, data.Length);
                        }
                        return memStream.ToArray ();
                }

                private void FileWriter(IEnumerable<byte[]> compressedBuffer) {
                        using (var stream = new FileStream("file.gz", FileMode.OpenOrCreate, FileAcce
                                foreach (var buffer in compressedBuffer) {
                                        stream.Write (buffer, 0, buffer.Length);
                                }
                        }
                }
        }
```

- Helper class - ThreadedList

```
        public class ThreadedList<T> : IEnumerable<T>
        {
                private readonly IEnumerable<T> _list;

                public ThreadedList (IEnumerable<T> list){
                        _list = list;
                }

                public IEnumerator<T> GetEnumerator ()
                {
                        return new ThreadedEnumerator<T>(_list.GetEnumerator ());
                }

                System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator ()
                {
                        return GetEnumerator ();
                }

                private class ThreadedEnumerator<S> : IEnumerator<S> {

                        private readonly IEnumerator<S> _enumerator;
                        private readonly Queue<S> _queue;
                        private const int _maxQueueSize = 10;
                        private readonly Thread _thread;
                        private volatile bool _keepGoing = true;
                        private volatile bool _finishedEnumerating = false;
                        private S _current;

                        public ThreadedEnumerator(IEnumerator<S> enumerator) {
                                _enumerator = enumerator;
                                _thread = new Thread(Enumerate);
                                _thread.Start();
                        }

                        private void Enumerate() {
                                while (_keepGoing) {
                                        if (_enumerator.MoveNext ()) {
                                                var current = _enumerator.Current;
                                                lock (_queue) {
                                                        while (_queue.Count > _maxQueueSize && _keepG
                                                                Monitor.Wait (_queue, 100);
                                                        }
                                                        if (_keepGoing) {
                                                                _queue.Enqueue (current);
                                                                Monitor.Pulse (_queue);
                                                        }
                                                }
                                        } else {
                                                break;
                                        }
                                }
                                _finishedEnumerating = true;
                        }

                        public bool MoveNext ()
                        {
                                lock (_queue) {
                                        while (!_finishedEnumerating && _queue.Count == 0) {
```

```
                                        Monitor.Wait (_queue, 100);
                                }
                                if (_queue.Count > 0) {
                                        _current = _queue.Dequeue ();
                                        Monitor.Pulse (_queue);
                                        return true;
                                } else {
                                        _current = default(S);
                                        return false;
                                }
                        }
                }

                public void Reset () {
                        lock (_queue) {
                                lock (_enumerator) {
                                        _enumerator.Reset ();
                                        _queue.Clear ();
                                }
                        }
                }

                object System.Collections.IEnumerator.Current {
                        get { return _current; }
                }

                public void Dispose () {
                        _keepGoing = false;
                        _thread.Join ();
                }

                public S Current {
                        get { return _current; }
                }
        }
}
```

1.3.4 Mutual Exclusion

- Mutual exclusion is a general problem that applies to both processes and threads.
- **Processes**
  - Occurs with processes that share resources such as shared memory, files, and other resources that must be updated atomically
  - When not otherwise shared, the address space of a process is protected against reads/writes by other processes
- **Threads**
  - Because threads share more resources such as having a shared process heap, there are more resources that need to be potentially protected
  - Because the address space is shared among threads in one process, cooperation and coordination is required for threads that read from and write to shared data structures
- When mutual exclusion is achieved, atomic operations on shared data structures are guaranteed to be atomic and not interrupted by other threads.

1.3.5 Tools for Achieving Mutual Exclusion

- **Mutex**
  - Has two operations: Lock() and Unlock()
  - Has two states: Locked or Unlocked
  - A lock can be acquired before entering a critical region and unlock can be called when leaving the critical region
  - If all critical regions are covered by a mutex, then mutual exclusion has been achieved and operations can be said to be atomic
- **Semaphore**
  - Has two operations: Up() and Down()
  - Has N states: a counter that has a value from 0 - N
  - Up() increases the value by 1
  - Down() decreases the value by 1
  - When the semaphore has a value > 0, then a thread of execution can enter the critical region
  - When the semaphore has a value = 0, then a thread is blocked
  - **The purpose of a semaphore is used to:**
    - Limit the number of threads that enter a critical region
    - Limit the number of items in a queue between two threads working in a pipeline processing pattern.
- **Monitor**
  - Has four operations: Lock(), Unlock(), Pulse(), Wait()
  - Allows for more complicated and user-coded conditions for entering critical regions
  - The locking semantics are more complicated for the simplest cases, but can express more complicated mutual exclusion cases in simpler ways than can semaphores or mutexes
- **Additional details may be found in the Operating Systems course**
  - Mutual Exclusion - http://osdi.cs.courseclouds.com/html/mutualexclusion.html
  - Deadlock - http://osdi.cs.courseclouds.com/html/deadlock.html

1.3.6 Mutex Example/Java

This code example shows how to implement a classic mutex, a.k.a. a Lock, in Java.

These examples come from http://hpjpc.googlecode.com by Christopher and Thiruvathukal.

1.3.7 Semaphore Example

This shows how to implement a counting semaphore in the Java programming language.

1.3.8 Barrier

This shows how to implement a barrier, which is a synchronization mechanism for awaiting a specified number of threads before processing can continue. Once all threads have arrived, processing can continue.

1.3.9 Deadlock - a classic problem

A classic problem in computer science and one that is often studied in operating systems to show the hazards of working with shared, synchronized state, is the *dining philosophers problem*. We won't describe the entire problem here but you can read http://en.wikipedia.org/wiki/Dining_philosophers_problem.

our "solution" has the following design:

- Fork: A class to represent the forks shared by adjacent philosophers at the table.
- Diner0: A class used to represent a philosopher. The philosopher does three things a philosopher normally does: think(), sleep(), and eat().
- Diners0: A class used to represent all of the diners seated at the table with their shared forks. This is where the concurrency takes place.

1.3.10 Fork

1.3.11 Diner0

1.3.12 Diners0

1.3.13 Diners1 - eliminating deadlock with resource enumeration

1.3.14 Execution - With Deadlock

If you have Java and Gradle installed on your computer, you can try these out!

Make sure you have the HPJPC source code:

```
hg clone https://bitbucket.org/loyolachicagocs_books/hpjpc-source-java
```

The following Gradle task in `build.gradle` shows how to run Diners0's `main()` method:

To build:

```
gradle build
```

To run:

```
gradle Diners0
```

If you run this, you will notice that deadlock ensues fairly quick. The diners get into a state where they are waiting on each others' forks in a cycle:

```
$ gradle Diners0
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:Diners0
tet4t 023et 12ett 0et40 e134e
et340 12ett 12ett 123et e1340
```

```
0234e e23e4 1tt40 t23et ett40
t23et 1ett0 12et0 t23et 1tt40
1t34e 12et0 1et40 12e30 t234e
12e30 1et40 tetet et3t4 1t3e4
1e240 1tte4 12tt0 t2ete t2tt0
11e3t et3t0 t234e e1340 11t40
1t340 0e24e tttet tt34e 12e3t
1t24e 0t3e4 tet4e 12340 12340
12340 12340 12340 12340 12340
12340 12340 12340 12340 12340
12340 12340 12340 12340 12340
12340 12340 12340 12340 12340
12340 12340 12340 12340 12340
12340 12340 12340 12340 12340
12340 12340 12340 12340 12340
12340 12340 12340 12340 12340
12340 12340 12340 12340 12340
12340 12340 12340 12340 12340
12340 12340 12340 12340 12340
12340 12340 12340 12340 12340
12340 12340 12340 12340 12340
```

1.3.15 Deadlock-Free Version

Diners1 has a similar Gradle task:

Run:

```
gradle Diners1
```

Output:

```
$ gradle Diners1
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:Diners1
ttttt 1t240 t2et4 1et4t tt2e4
1e2et 1et4t e13e0 tt3e4 ettt0
t2te0 0e24e 1ettt e1e3t t1te4
0tt4e 1etet e13tt tt24e 1t3et
tettt 0ttet ete3t tt33e 0et4e
1ete0 01t3t 0tt3e 1e240 te2te
e1et0 1e2et 02e3e t1t3t 1t3tt
02ete 1et4t e13et et33t 02tte
1ett0 et3t0 ete30 t2e3e et3e0
0et4e ettt0 0e2e4 01t4e 1e2et
...
12e30 tet3e 1etet 0ttt0 0etet
1et4t e2tt4 tt3e4 0t3et 12et0
1ett0 e1tet 12e30 1tttt etet0
tettt 1e2t0 0t3e4 tettt ttttt
023e4 ttttt 023e4 1e2d0 e13d0
02ed4 e2edt 1etd0 et3d0 1tedt
02ede 0etde 1etd0 t2tdt t2ede
01tde et2d0 112dt tedde tedd4
```

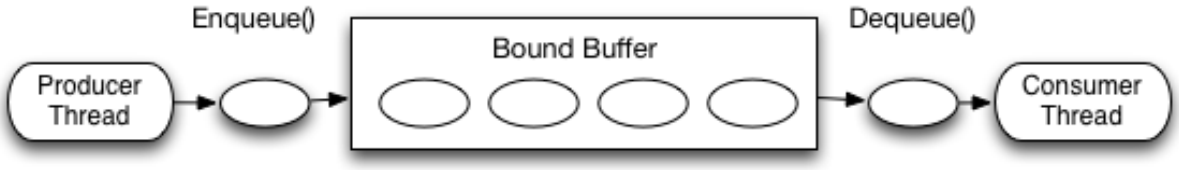
```
tedde 02dd0 1edd0 etdd0 1tddt
1eddt 1eddt 01dde 0tdd0 t2dde
t2ddt eddd4 tddde tddd4 tdddt
0ddde eddd0 tdddt 0ddde eddd0
tddd4 eddd0 0ddde 0ddde eddd0
eddd0 0ddde 0ddde tddd4 0dddt
eddd0 tddd4 1dddd 0dddd 1dddd
tdddd tdddd ddddd
BUILD SUCCESSFUL

Total time: 18.426 secs
```

The diners, as desired, end up finishing their meal after some time.

We assume they have moved over to the bar or found a nice place to have dessert.

1.3.16 Common Data Structures in Concurrent Programming

- **Bound Buffer**
  - Makes use of a mutex and semaphore internally
  - Defines a maximum number of items that exist in the bound buffer's queue.
  - Has two operations: Enqueue() and Dequeue()
  - Enqueue() - enqueues items in the data structure. If the enqueue operation would cause the bound buffer to exceed the maximum, the Enqueue() call will block until another thread dequeues at least one item.
  - Dequeue() - dequeues an item from the data structure. If there are zero items in the queue, Dequeue() will block until another thread enqueues an item in the data structure
  - Bound buffers are used to make sure that when one thread is producing work for a second thread, that if one thread is faster or slower than the other, that they appropriately wait to some extent for each other.



- Example Bound Buffer

```
1        public class BoundBuffer<T> {
2                private readonly Semaphore _full;
3                private readonly Semaphore _empty;
4                private readonly Semaphore _lock;
5                private readonly Queue<T> _queue;
6
7                public BoundBuffer (int maxCount) {
8                        _empty = new Semaphore (maxCount, maxCount);
9                        _full = new Semaphore (0, maxCount);
10                       _lock = new Semaphore (1, 1);
11                       _queue = new Queue<T> ();
12               }
```

```
13
14                  public void Enqueue(T item) {
15                          _empty.WaitOne ();
16                          _lock.WaitOne ();
17                          _queue.Enqueue (item);
18                          _lock.Release (1);
19                          _full.Release (1);
20                  }
21
22                  public T Dequeue() {
23                          _full.WaitOne ();
24                          _lock.WaitOne ();
25                          var item = _queue.Dequeue ();
26                          _lock.Release(1);
27                          _empty.Release(1);
28                          return item;
29                  }
30          }
```

1.3.17 Design Considerations

- **Threading requires the support of the operating system - a threading library / package is needed**
  - In Windows, this is a part of the Windows SDK and .NET Framework
  - In Linux and Mac OSX, PThreads provides threading
- **Thread usage and creation**
  - Threads can be started and stopped on demand or a thread pool can be used
  - **Starting threads dynamically:**
    - Has some cost associated with asking the OS to create and schedule the thread
    - It can be architecturally challenging to maintain an appropriate number of threads across software components
    - This is overall the most simple approach
  - **Thread Pools**
    - The number of threads can be defined at compile time or when the program is first launched
    - Instead of creating a new thread, the program acquires a thread and passes a function pointer to the thread to execute
    - When the given task is completed, the thread is returned to the pool.
    - This approach does not have the overhead of creating / destroying threads as threads are reused.
    - This approach often requires library support or some additional code.
- **The total number of threads**
  - Having several hundred threads on a system with an order of magnitude fewer cores can cause you to run into trouble.
  - If a majority of those threads are runnable, then the program will spend most of its time context switching between those threads rather than actually getting work done.
  - If such a system is dynamically starting and stopping threads, then the program will most likely spend most of its time creating and destroying threads.

1.3.18 Kernel Threads vs User Mode Threads

- **There are two types of threads:**
  - **Kernel Threads** -Supported by modern operating systems -Scheduled by the operating system
  - **User Threads** -Supported by almost everything -Scheduled by the process
- **Context switching:**
  - Kernel threads have a higher overhead because the scheduler must be invoked and there might be a time lag before a runnable thread is actually executed.
  - Kernel threads often perform very well because the operating system has more information about the resource state of the computer and can make better global scheduling decisions than can a program
  - User-mode threads can context switch with fewer overall operations, but scheduling them is guesswork.
  - User mode threads can be created more rapidly because new stacks and scheduler entries do not need to be created by the operating system
- **Where are user-mode threads used?**
  - In systems without kernel mode threads
  - When the number of threads a system needs is in the hundreds or thousands (user-mode threads scale better in these scenarios)
- **Where are kernel-mode threads used?**
  - When the number of threads is not very high (less than 10 per core)
  - When blocking calls are involved (user-mode thread libraries usually have separate I/O libraries)

1.3.19 Concurrent File Copy Example

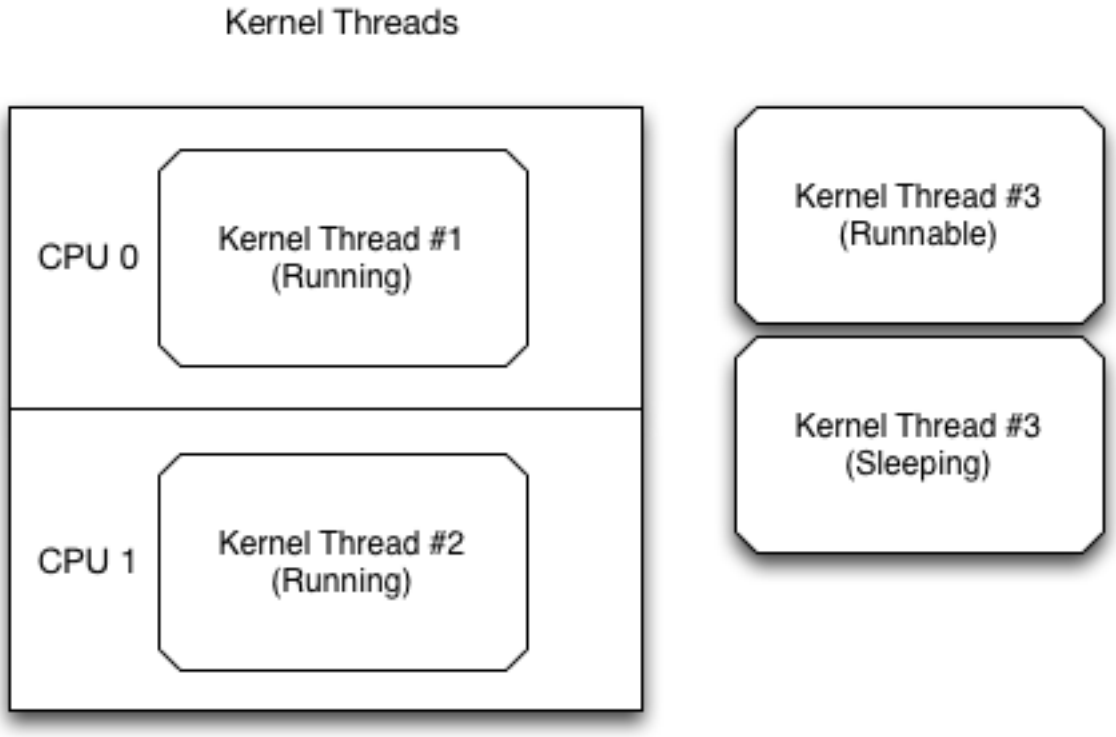
- FileCopy0: The sequential version
- FileCopy1: The concurrent version

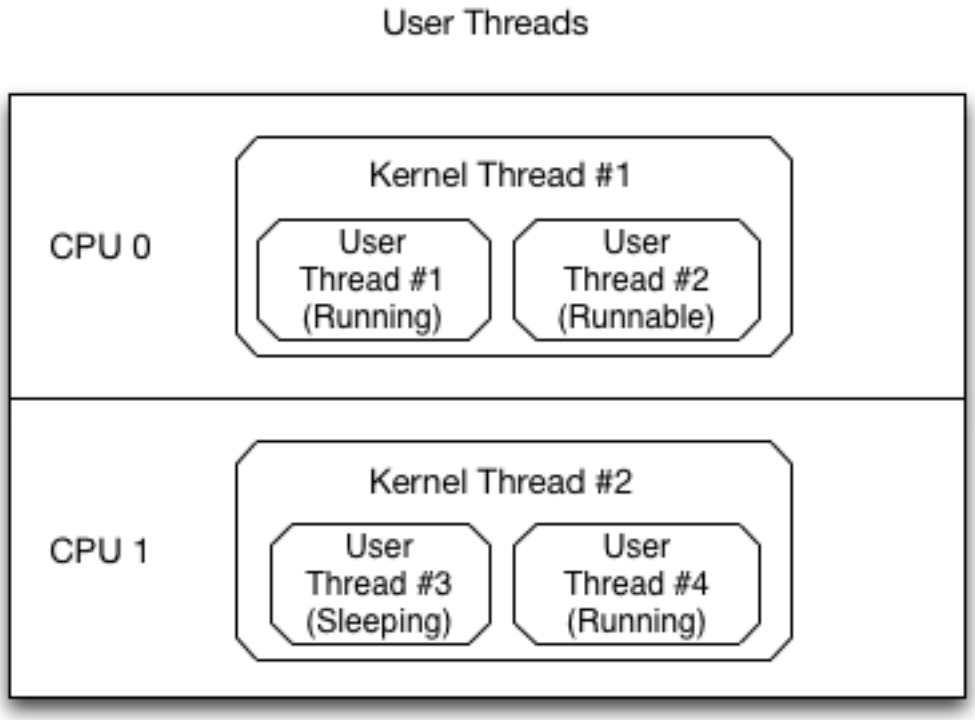
1.3.20 Sequential File Copy

1.3.21 Concurrent File Copy Organization

Quick overview of the various classes:

- Pool: Maintains a list of buffers that can be used for allocating/freeing blocks of data without triggering new (or dispose) repeatedly.
- Buffer: Used as a shared object for reading and writing blocks of data (via the FileCopyReader1 and FileCopyWriter1 classes)
- BufferQueue: Used to queue up blocks as they are read or written. This allows for varying speeds of reader and writer, subject to the number of blocks available in the Pool.
- FileCopyReader1: Used to run the reader thread.
- FileCopyWriter1: Used to run the writer thread.
- FileCopy1: Used to act as a drop in replacement for FileCopy0. Sets up the reader and writer threads and then joins with both when the reading/writing are completed.


Kernel Threads
CPU 0
Kernel Thread #1
(Running)
CPU 1
Kernel Thread #2
(Running)
Kernel Thread #3
(Runnable)
Kernel Thread #3
(Sleeping)


User Threads
CPU 0
Kernel Thread #1
User
Thread #1
(Running)
User
Thread #2
(Runnable)
CPU 1
Kernel Thread #2
User
Thread #3
(Sleeping)
User
Thread #4
(Running)

1.3.22 Execution

You can run FileCopy0 and FileCopy1 by using the corresponding Gradle tasks.

As shown, there are two properties you can set: `fc0_src` and `fc0_dest`:

```
gradle FileCopy0 -Pfc0_src=inputFile -Pfc0_dest=outputFile
```

You can also run FileCopy1 (the same parameter names are used):

```
gradle FileCopy1 -Pfc0_src=inputFile -Pfc0_dest=outputFile
```

1.4 Distributed Systems and Storage

1.4.1 Types of Non-Local Storage

1. SANs - Storage Area Networks
2. NASs - Network Area Storage
3. NFS/CIFS - Network File System / Common Internet File System
4. Cloud Storage / Sync Services
5. Distributed File Systems
6. Parallel File Systems

1.4.2 How Do We Evaluate Storage Systems?

- Latency - How long does a single operation of the smallest fundamental unit take to complete.
- Throughput - How many bytes of data per second can we read or write
- Parallel scaling - When several requests are issued at the same, or nearly the same time, how are latency and throughput affected.
- Resilience to failure of storage components - How many and which types of storage hardware failures can be tolerated without the system availability being interrupted and how is performance affected.
- Resilience to failure of network components - If a network is partitioned or fails, does the storage system remain consistent and how are partially complete operations handled
- Semantics - What capabilities are offered to clients? Can existing files be re-written? Are there folders? Is random access possible? How are files locked and shared? Are there any transactional semantics?
- Location Transparency

1.5 Storage Devices

- Types of permanent devices:
  - Magnetic - hard disk, tape, floppy disk
  - Optical - CD/DVD/Blu-Ray, Laser Disc, Paper, Punch Cards, Photo Film
  - Solid State - CMOS, NAND based flash, battery backed dynamic memory

- Types of transient devices:
  - RAM, Processor Caches

1.5.1 Storage and Failure

- An excellent paper about hard disk failure rates at Google is available here:
- https://www.usenix.org/legacy/event/fast07/tech/full_papers/pinheiro/pinheiro.pdf
- At Google, the failure rates of disks of a given age are:
  - 3-months - 2.5
  - 6-months - 2.0
  - 1-year - 2
  - 2-year - 8
  - 3-year - 8.5

1.5.2 Maximizing Availability - RAID

- To prevent the loss of availability of data, the use of RAID (Redundant Array of Inexpensive Disks) allows for redundant copies of data to be stored.
- Common RAID levels are:
  - RAID 0 - splits data across disks. Increases disk space and provides no redundancy. 2 or more disks are needed.
  - RAID 1 - creates an exact copy of data on two or more disks.
  - RAID 5/6 - splits data across disks. Uses one or more disks for parity. This allows 1-K out of N disks to fail and allow the data of any lost disk to be recovered. 3 or more disks are needed.

1.5.3 RAID

- RAID has three common implementation approaches:
  - Complete hardware implementation - a disk controller or expansion card implements RAID. Several disks are connected to this controller and it is presented to the operating system as a single storage device. Often have reliability guarantees.
  - Partial hardware implementation - Same as the complete hardware implementation, except parity calculations, and buffering are delegated to the host CPU and memory. Don't often have reliability guarantees.
  - Software implementation - The operating system itself manages several disks and presents to the file-system layer a single storage device.

1.5.4 RAID - 0

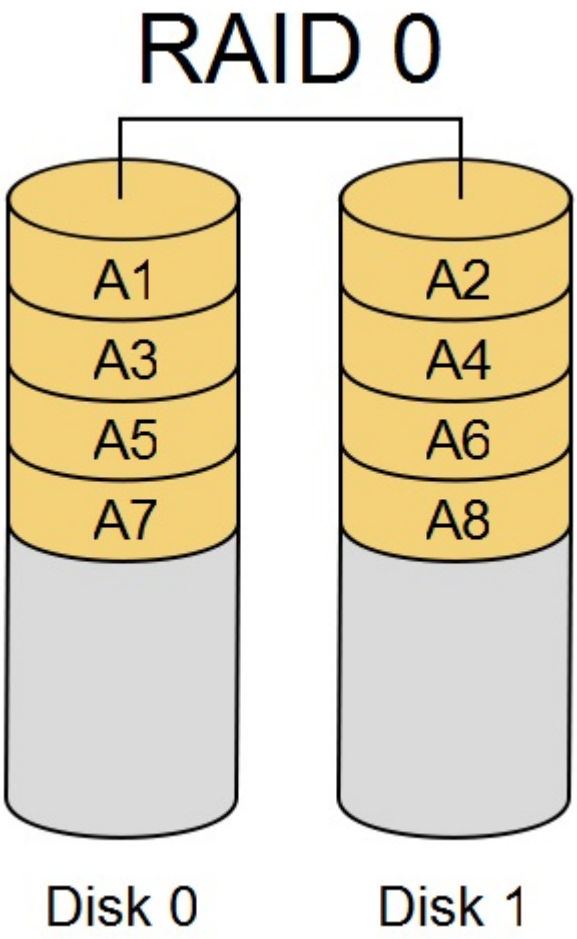


Fig. 1.11: image

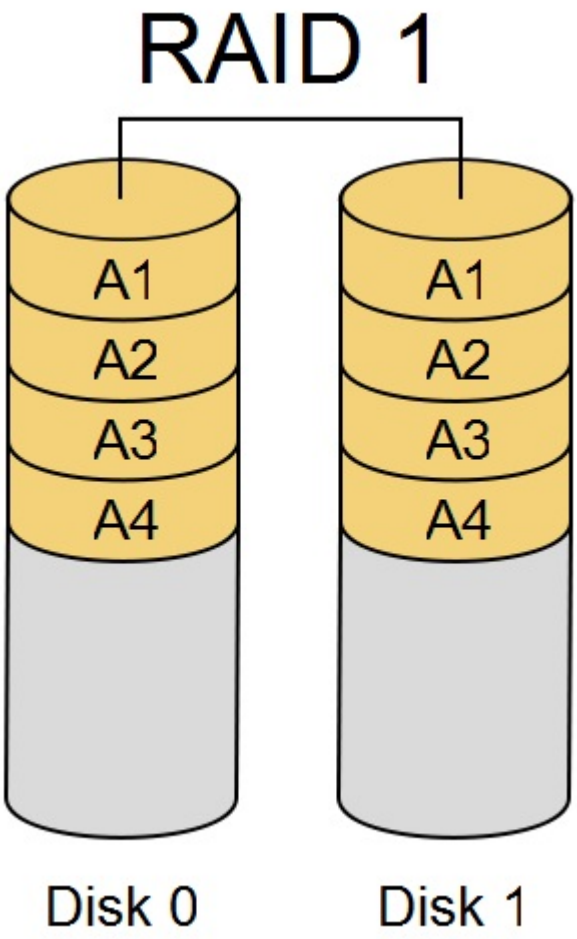


Fig. 1.12: image

1.5.5 RAID - 1

1.5.6 RAID - 5

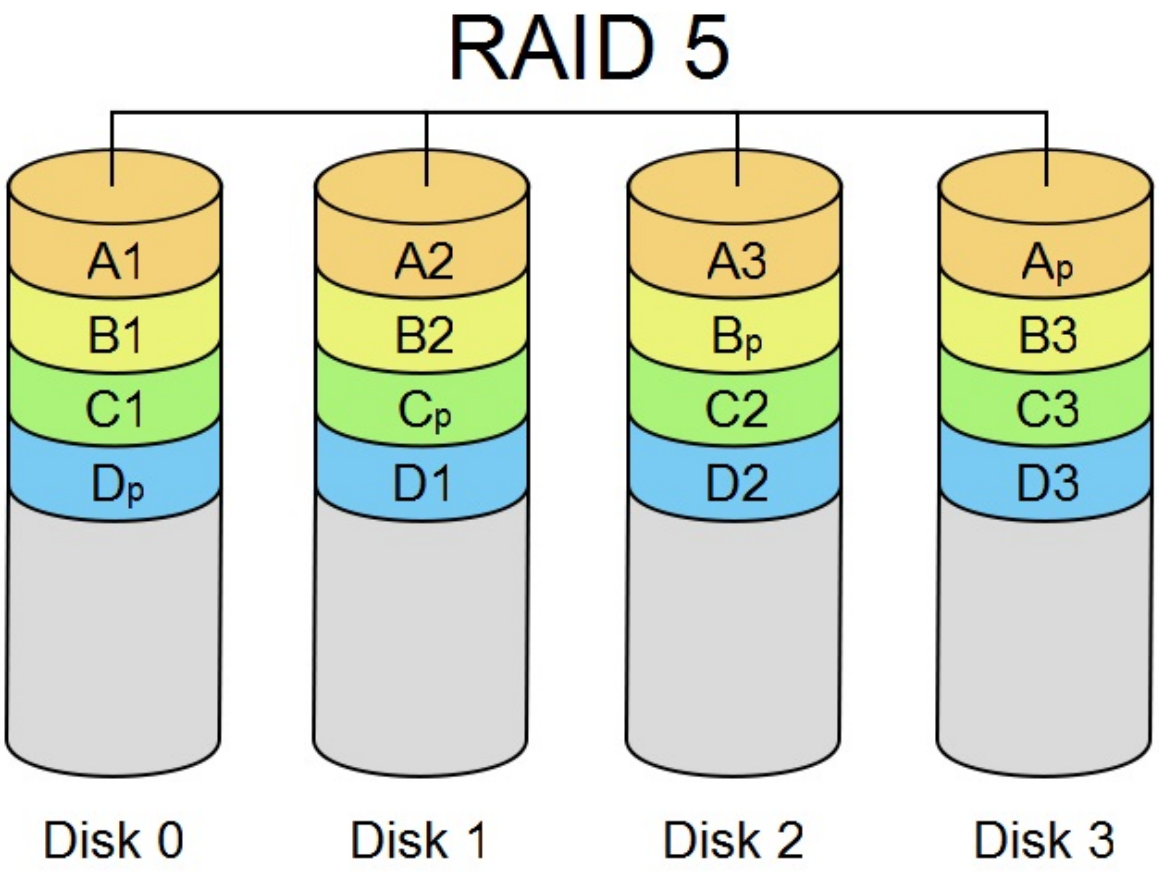


Fig. 1.13: image

1.6 Local Storage

- At the basis of almost any distributed system are the factors involved in local storage systems.
- The problems presented in local storage are simpler and less composed than in their distributed counter-parts.

1.6.1 Implementing Files and Folders

- How files and folders are implemented in a storage medium can greatly depend upon the physical characteristics and capabilities of that medium.

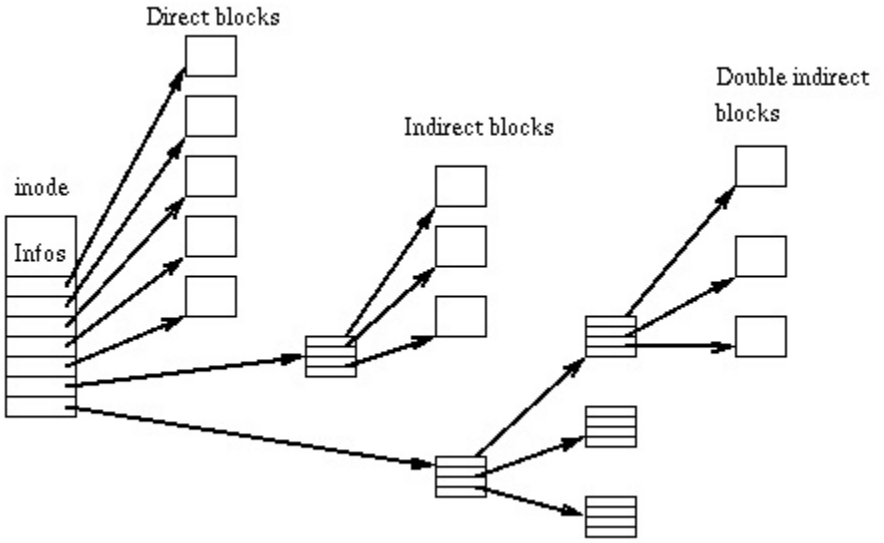
- For example, on tape-drives, CD/DVD/Blu-Ray, or write-once media, files and folders are stored contiguously with no fragmentation. All of the information about the filesystem can be held in a TOC (Table Of Contents).
- For filesystems with files that have a finite lifetime, such as on flash media, hard disks, SSDs, and others, the layout of files and folders must be maintained in a more complex way.
- Among these more advanced methods are linked lists and i-nodes.
- To manage free-space, objects like bit-maps and linked lists are possibilities.

1.6.2 Inodes

- inodes are the fundamental structures of a UNIX filesystem
- inodes have the following attributes:
  - File Ownership - user, group
  - File Mode - rwx bits for each of user, group, and others
  - Last access and modified timestamps
  - File size in bytes
  - Device id
  - Pointers to blocks on the storage device for the file or folder's contents

1.6.3 Inodes - Indirect Blocks

- The strategy of using indirect, double indirect, and even triple indirect blocks is a very successful implementation strategy
- This approach is used by ext2 / ext3 / ext4 in Linux.



1.6.4 Block Caches

- To improve the performance of a filesystem, and to make disk scheduling algorithms more realizable, most operating systems implement some kind of block cache.
- The block cache allows for read-ahead and write-behind. It also allows for lower latency I/O operations.
- With a block cache, the write() system call for instance only needs to complete modifications to the cache before returning. The operating system can complete the operation on disk in a background thread.
- Without this cache, the system call would not be able to return until the write had been committed to disk.
- Important parameters of any block cache are:
  - The size of the cache in physical memory
  - The delay before committing 'dirty' items in the cache to disk
- The larger the cache, the better the filesystem will likely perform, but this can come at the cost of available memory for programs.
- The larger the delay before writing items to the disk, the better the disk allocation and scheduling decisions the operating system can make.
- The shorter the delay before writing to disk, the greater the guarantee in the presence of failure that modifications will be persisted to disk.

1.6.5 Folders and Path Traversal

- In all but the most simple filesystems, there is a concept of a folder and a path.
- In UNIX operating systems, folder entries are held within inodes that have the filetype in the mode set to type directory.
- The contents of the inode then are a list of filenames and pointers to the inodes of those files and/or folders.
- Resolving paths involve accessing a root folder, and accessing each folder recursively until reaching a file or finding the folder to be invalid.
- An example of path traversal. When traversing paths, the path may cross into different filesystems.

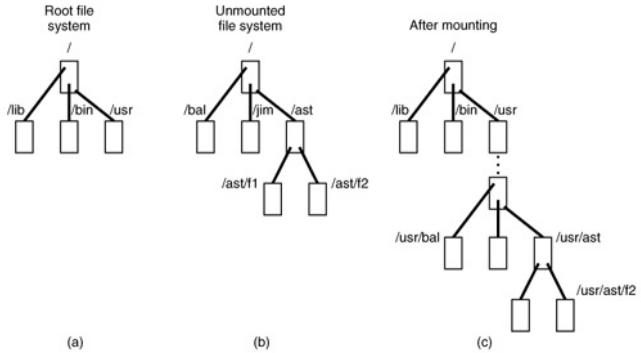


Figure 5-38. (a) Root file system. (b) An unmounted file system. (c) The result of mounting the file system of (b) on */usr/*.

1.6.6 Virtual Filesystems / VFS

- Aside from files and folders there are other things like named pipes, domain sockets, symbolic and hard links that need to be handled by the filesystem.
- Rather than have the semantics of these implemented in each filesystem implementation, many OS architectures include a virtual filesystem or VFS.
- The VFS stands between the OS kernel and the filesystem implementation.

1.6.7 Virtual Filesystems / VFS

- The VFS can help adapt both foreign filesystems (such as VFAT) by producing a contract that these implementations can adapt to.
- The VFS can also help reduce code duplication between FS implementations by providing common structures and handling shared behavior:
  - Path traversal
  - Handling named pipes, domain sockets, etc...
  - Managing file handles and file locking
  - Structures and functions for the block cache.
  - Structures and functions for accessing storage devices

1.6.8 Virtual Filesystems and Stacking

- In some VFS implementations it is possible to stack filesystems on top of each other.
- A great example of this in Linux is UMSDOS: the base VFAT filesystem does not have support for users, groups, security or extended attributes. By creating special files on VFAT and then hiding them, UMSDOS can adapt VFAT to be a UNIX-like filesystem
- Another great example of this is UnionFS. It allows two filesystems to be transparently overlaid.

1.7 Distributed Filesystems

- **Flat file service**
  - implements operations on the contents of file
  - UFID (Unique File Ids) used to refer to files
  - new UFID assigned when file created
- **Directory service**
  - provides mapping between text names and UFIDs
  - Functions to create, update.. directories
- **Client module**
  - runs on client computer
  - provides APIs to access files

  - holds information about network location of file server and directory server
  - sometimes caching at client

1.7.1 File Service Model

- **Upload/download model**
  - read/write file operations
  - entire file transferred to client
  - requires space on client
  - Products like SkyDrive and DropBox work like this
- **Remote Access Interface**
  - **large number of operations**
    - seek, changing file attributes, read/write part of file…
    - does not require space (as much) on client

1.7.2 Directory Service

- **Key issue for distributed file system**
  - whether all clients have the same VIEW of the directory hierarchy

1.7.3 Naming Transparency

- **Location Transparency**
  - path names give no hint as to where the files are located
  - e.g., /server1/dir1/dir2/X indicates X located on server1 but NOT where server1 is located
  - Problems? If X needs to be moved to another server (e.g., due to space) - say server2 - programs with strings built in will not work!
- **Location Independence**
  - files can be moved without changing their names
- **Three common approaches to file and directory naming**
  - Machine + path naming, such as /machine/path or machine:path (location dependent)
  - Mounting remote file systems onto the local file hierarchy (location dependent)
  - A single name space that looks the same on all machines (location independent)

1.7.4 File Sharing Semantics

- When files are shared (and one or more write) what are the semantics?

1.7.5 UNIX Semantics

- A read is always provided with the last write (system enforces absolute time ordering)
- **UNIX semantic can be achieved by**
  - read/write going to server
  - no caching of files
  - sequential processing by server
  - BUT in distributed systems, this may perform poorly!
- **How to improve performance?**
  - requires caching
- **Modify Semantics?**
  - "changes to an open file are initially visible only to the process that modified the file. When file closes, changed become visible to others"
  - Called Session Semantics

1.7.6 More Semantics

- **Q What is the result of multiple (simultaneous) updates of cached file?**
  1. final result depends on who closed last!
  2. one of the results, but which one it is can not be specified (easier to implement)
- **Immutable Files**
  - can only create and read files
  - can replace existing file atomically
  - to modify a file, create new one and replace
  - what if two try to replace the same file?
  - what if one is reading while another tries to replace?

1.7.7 Distributed File System Implementation

- **Need to understand file usage (so that)**
  - implement common operations well
  - achieve efficiency
- Satyanarayan (CMU) of file usage pattern on UNIX

1.7.8 System Structure

- **How should the system be organized?**
  - are clients and server different?
  - how are file and directory services organized?

    - **caching/no caching**
        - server
        - client
    - how are updates handled?
    - sharing semantics?
    - stateful versus stateless

1.7.9 Directory Service

- **Separate**
    - (-) requires going to directory servers to map symbolic names onto binary names
    - (+) functions are unrelated (e.g., implement DOS directory server and UNIX server- both use same file server
    - (+) simpler
    - requires more communication
- Lookup

1.7.10 Stateless versus Stateful

- **Stateless advantages**
    - Fault tolerance
    - No OPEN/CLOSE calls needed
    - No server space wasted on tables
    - No limits on number of open files
    - No problem if client crashes
- **For example,**
    - each request self contained
    - if server crashes - no information lost

1.7.11 Caching

- **One of the most important design considerations**
    - impacts performance
    - If caching – how should it be done?

1.7.12 Caching - Server

- **Server Disk**
  - (+) most space
  - (+) one copy of each file
  - (+) no consistency problem
  - **(-) performance**
    - each access requires disk access –> server memory –> network –> client memory
- **Server Memory**
  - keep MRU files in server's memory
  - If request satisfied from cache ==> no disk transfer BUT still network transfer
  - 17. **Unit of caching? Whole files**
    - (+) high speed transfer
    - (-) too much memory
  - Blocks + better use of space
  - 17. **What to replace when cache full?**
    - LRU

1.7.13 Caching - Client

- Client Caching
- **Disk**
  - slower
  - more space
- **Memory**
  - less space
  - faster
- Where to cache?
- **User Address Space**
  - cache managed by system call library
  - library keeps most heavily used files
  - when process exits - written back to server
  - (+) simple
  - (+) low overhead
  - (-) effective if file repeatedly used
- **Kernel**
  - (-) kernel needed in all cases)even for a cache hit)

  - (+) cache survives beyond process (e.g., two pass compiler - file from first pass available in cache)
  - (+) kernel free of file system
  - (+) more flexible
  - **little control over memory space allocation**
    - e.g., virtual memory may result in disk operation even if cache hit

1.7.14 Client - Cache Consistency

- **client caching introduces inconsistency**
  - one or more writers and multiple readers
- **Write-thru**
  - similar to between processor cache and memory
  - when a block modified - immediately sent to server (also kept in cache)
- **problem**
  - **client on machine 1 reds file**
    - modify file (server updated)
  - **client on machine 2 reads and modifies files**
    - server updated
  - **another client on machine 1 reads file**
    - gets local copy (which is stale)
- **solution: write-thru**
  - cache manager checks with the server before providing file to client
  - **If local copy upto-date**
    - provide to client
  - Else get from server
  - RPC for check is not as expensive as file access
- **Performance problems**
  - read is fine
  - each write generates network traffic (very expensive)
  - compromise - periodic updates (say 30 sec) of writes
  - collected and sent to server
  - eliminates writing of many scratch files completely (which otherwise would be written)
- Note- semantics have changed for delayed writes

1.7.15 Client - Cache Consistency - Other Options

- **Write-on-Close**
  - session semantics
  - **wait (delay - say 30 sec) after close to see if file deleted**
    - in that case write eliminated
- **Centralized Control**
  - File server keeps track of who has file and in what mode
  - if new request for read - server checks to see if file opened read/write
  - if read - grant request
  - if write - deny request
  - when file closed - inform others
  - Many variations possible

1.7.16 Replication

- **Multiple copies of files for**
  - increased reliability so no data is lost
  - increased availability when one server is down
  - improved performance through division of load

1.7.17 Replication - Update Protocols

- **send update to each file in sequence**
  - problem - if process updating crashes in the middle ==> inconsistent copies
- **Primary Copy Replication**
  - one server designed as primary
  - primary updated (changes made locally by primary server)
  - primary server updates secondary copies
  - reads can be done from any copy
  - **to guard against primary copy failure**
    - updates first stored on stable storage
  - But if primary copy down - No update can be made!!

1.7.18 Replication - Voting Algorithm

- Requires clients to acquire permission of multiple servers before reading/writing file
- File replicated on N servers - to update client needs to contact majority , N/2 + 1 servers
- File changed and new version no assigned

- **To read - client contacts N/2 + 1 servers**
  - will always get the latest version

1.7.19 Replication - General Quorum Algorithm

- No of replicas - N
- Read Quorum - Nr
- Write Quorum - Nw
- Constraints Nr + Nw > N
- Read/write requires participation of the corresponding quorum

1.8 Case Study - SUN NFS

- **NFS - Network File System**
  - designed to allow an arbitrary collection of clients and servers to share a common file system
- **Design Goals**
  - heterogeneity
  - access transparency
  - local and remote accesses look the same - e.g., normal UNIX system calls
  - failure transparency
  - stateless
  - idempotent
  - performance transparency
  - client caching
  - server caching
- **Location Transparency**
  - client establishes file name space by adding remote file systems to local name space
  - file system exported by servers (node holding it)
  - file system remote-mounted by client
- **Not Supported in Design Goals**
  - **Replication transparency**
    - separate service for replication (NIS)
  - **Concurrency**
    - Naïve locking
  - **Scalability**
    - limited
    - originally designed to support 5-10 clients

1.8.1 SUN NFS - Implementation

- **VFS Layer**
  - maintains table with one entry for each open file
  - entry called v-node (indicates whether local or remote)
  - v-node points to I-node (for local files) and r-node (for remote files)
- **Typical Operation**
  - **Read**
    - locate v-node
    - determine local or remote
    - transfer occurs in 8K (normally) byte blocks
    - automatic prefetching (read-ahead) of next block
  - **Write**
    - writes not immediately written to server
    - 8K bytes collected before writing
- **Caching**
  - server caches (to reduce disk accesses)
  - **client maintains cache for**
    - for file attributes (I-nodes)
    - for file data
  - **cache block consistency problems**
    - with each cache block is a timer
    - entry discarded when timer expired
    - when file opened- server checked for last modification of file
  - UNIX semantics not completely enforced

1.9 Directories and LDAP

1.9.1 What is a Directory?

- Yellow pages
- Personal phone/address book
- Mail order catalogs
- Library catalog cards
- TV Guides
- Your organization's mailing list
- Netscape's HTML directory

1.9.2 What is NOT a Directory?

- Database: more costly, heavy-duty transaction support, frequent writes
- File system: allows partial retrieval, random access of data
- Web servers: delivers big image files, provides web application development platform
- FTP servers: can't do search, not attribute-based information model
- DNS servers: not extensible, no updates

1.9.3 Types

Types of (Network) Directories

- NOS-based: MS Active Directory (with LDAP core), Novell NDS
- Application-specific: Lotus Notes Address Book, MS Exchange Directory
- Purpose-specific: DNS
- General-purpose, standards-based: LDAP, X.500 directories

1.9.4 Characteristics of Directories

Characteristics of Directories - Good at storing pointers to large data, not the large data - Attribute-based information model - High read-to-write ratio, unlike databases, file systems, or web browsers - High search capability - Standards-based access (for some)

1.9.5 Evolution of Directories

FIGURE

Directory Services and Organizations

The Nerve Center of an Organization's Infrastructure?

- Naming: who/what are there?
- Location: where are they?
- Security: protect against unauthorized access and tempering
- Management: personnel, decision- making
- Resource: monitor load and usage
- Extensibility: grow with the organization

1.9.6 So what is LDAP, exactly?

- LDAP: Lightweight Directory Access Protocol
- Preceded by the two X.500-related protocols: DAS, DIXIE (front ends)
- "Lightweight": simplified encoding methods, runs directly on commonly available TCP/IP

- "Heavyweight":X.500DAPuses complex encoding methods, runs on rare OSI network protocol stack
- Simplified implementation of clients and servers by eliminating infrequent and redundant DAP features/operations
- Data elements represented as simple text strings, while messages wrapped in binary encoding for efficiency
- Uses a subset of the X.500 encoding rules to further simplify implementation
- Simplified transport: no need for OSI, runs directly over TCP.
- Supports both IPv4 and IPv6.

1.9.7 Early LDAP Implementations

FIGURE

1.9.8 SLAPD: Stand-alone LDAP Daemon

- Multi-platform LDAP directory server
- Flexible customization
- Support for both LDAPv2 and LDAPv3
- Support for both IPv4 and IPv6
- Simple Authentication and Security Layer
- Transport Layer Security through SSL
- Generic modules API: covers development for front-end communication with LDAP client, and back-end database operations with Perl, Shell, SQL, TCL, and Python

1.9.9 SLAPD: Continued

- Access control based on LDAP authorization information, IP address, domain name, etc.
- Support for Unicode and language tags
- Choice of various backend databases
- A multi-threaded **slapd** process can handle all incoming requests => reduced system overhead => higher performance
- Replication of data using single-master multiple-slave scheme for high-volume environments
- Highly configurable via a single configuration file that "does it all"

1.9.10 LDAP Client-Server Exchange Protocol

FIGURE

1.9.11 Example LDAP Usage

- A directory service that enables a user to locate remote resources ANYWHERE on a distributed network.
- A dynamic system that provides/fetches resources based on individual user's queries.
- A model that would utilize and manage the existing system in a more organized way.
- Examples:"OmniPrint" service from anywhere on the network, location and availability of a coffeemaker!
- A robust and extensible client-server model
- Application needs: data elements, service performance (latency, throughput, e.g., 480K searches per hour)
- User needs: accuracy, privacy, up-to-date, completeness, security, balance for all users
- Extensibility: Must be able to extend to distributed and replicated models
- Platforms supported: Should accommodate heterogeneous platforms

1.9.12 Schemas

- Similar to databases, needed for integrity and quality
- A set of rules that determines what can be stored in a directory service
- A set of rules that defines how directory servers and clients should treat information during a directory operation
- Each entity (called "attribute") has its own object identifier (called an *oid*)
- Reduce unnecessary data duplication resulted from some directory-enabled applications

1.9.13 attributes and objectclasses

The following shows how to create your own schema in LDAP (for our example fictitious domain):

```
description ATTRIBUTE ::= {
  WITH SYNTAX DirectoryString {1024}
  EQUALITY MATCHING RULE caseIgnoreMatch
  SUBSTRINGS MATCHING RULE caseIgnoreSubstringsMatch ID 2.5.4.13
}

objectclass printer
  requires cn
  allows description, pagesPerMinute, languages

objectclass networkDevice
  requires ipaddress
  allows cn, connectionSpeed
```

Note: the term objectclass is not the same terminology from object-oriented programming. In LDAP, an objectclass defines a schema-aware data object but does not define methods (functions) as in OOP.

1.9.14 Namespaces

- Means by which information in the directory will be named and referenced, similar to a pointer or label (or index).
- Namespace can be of any topology, e.g. tree, star, triangular, or linear. LDAP supports trees innately.

- Concept of DN and its components: CN, C, ST, L, O, OU, STREET, DC, UID
- Naming scheme could be internet-based or traditional, based on organizational needs

Here's an example of a DN:

```
ou=cpdc,ou=ece,ou=northwestern,ou=edu, c=evanston,st=illinois,c=us
```

Traditional (Internet-style) naming:

```
CPDC.ECE.McCormick.Northwestern.Evanston.IL.US
```

Which is better?

1.9.15 LDAP Tree Topology

FIGURE to show the distinguished name concept.

1.9.16 Network Architecture

FIGURE

1.9.17 Distributed LDAP Server Model

FIGURE

1.9.18 LDIF Example

```
dn: uid=lt412-p3,ou=People,dc=cs,dc=luc,dc=edu
uid: lt412-p3
cn: LT 412 P3 Lab
givenName: LT 412 P3
sn: Lab
objectClass: person
objectClass: organizationalPerson
objectClass: inetOrgPerson
objectClass: posixAccount
objectClass: top
objectClass: shadowAccount
userPassword: {crypt}$1$/I5v6ig6$aDHu3Idj8i98kb9XVHlvq0
shadowLastChange: 12695
shadowMax: 99999
shadowWarning: 7
loginShell: /bin/bash
uidNumber: 1012
gidNumber: 250
homeDirectory: /homes/users/lt412-p3
gecos: LT 412 P3 Lab
```

1.9.19 Acknowledgements

The notes in this lecture are based on a presentation co-authored with Dr. Thiruvathukal's former students in Distributed Systems (at Northwestern University), Steve Chiu and Jay Pisharath.

1.9.20 References

- http://openldap.org
- T. Howes et. al., *Understanding and Deploying LDAP Directory Services*, MacMillan Technical Publishing, 1999
- LDAP bindings are provided in many languages, such as Python and Java. OpenLDAP provides C bindings.

1.10 Continuous Integration

1.10.1 What Does Continuous Integration Do?

- To support incremental, agile development for one or more developers
- Provides notification of success or failure of forward integration as rapidly as possible. This is especially important on larger teams.
- Performs validation of the product by running unit tests, and automated user interface tests
- Can perform scheduled test or production environment deployments.

1.10.2 How Often is Continuous Integration Run?

- **Continuous Integration can be run as:**
  - Batches - once one build and verification is complete, another immediately starts (grouping more than once change, potentially)
  - Upon changes - build and validate the product at every change
  - Iteration builds - every day, every week, or some other short period, a distributable product with an installer or package is created. This can be used for manual testing by a test team or by customers.

1.10.3 What are the Bits and Pieces of Continuous Integration?

- **Build Controllers:**
  - A service that coordinates build requests and build operations
  - Manages resources such as artifact repositories, build folders, and individual build nodes.
  - One or more of these may exist depending upon the product involved.
  - Usually a very I/O and memory bound service
- **Build nodes:**
  - Answer requests from the build controller (the work horse of continuous integration)
  - Pulls source code, performs compilation, executes automated tests
  - Some environments have as few as 2-3 build nodes, some have hundreds
  - Usually a CPU and I/O bound service
- **Artifact Database:**
  - Stores the N most recent continuous integration builds

  - Stores external 3rd party libraries used in building various versions of the product
  - Stores several versions of released builds of individual modules of the product (in the case of component or plugin architectures)
  - Stores installers for released or testable products
  - Typically has dozens of TB of storage on a fast SAN
- **Deployment of Continuous Integration:**
  - Build controllers typically have fairly high resource utilization
  - Build nodes typically do not have high resource utilization
  - The process of compiling and running unit tests typically does not fully utilize the power of a build node
  - So, a common practice is to virtualize hosts that run build nodes and place at run at least two hosts per build node
  - It is uncommon to allow more than one build task to occur simultaneously on a build node
  - Build nodes are commonly re-imaged every day - every few days.
  - Many continuous integration products are capable of running in the cloud with dynamic resource utilization for the build nodes. Both Team City and Team Foundation Server are well known for this

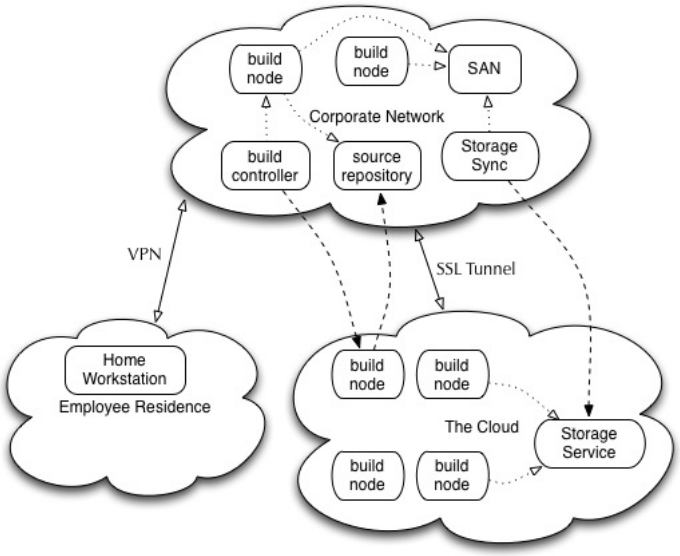


Fig. 1.14: A Possible Corporate Deployment of a Continuous Integration System

1.10.4 Continuous Integration Products / Vendors

- Many exist, but these are what you see out there in the wild:
- Jet Brains - Team City
- Microsoft - Team Foundation Server
- Borland - Silk

- IBM - Rational Suite
- Apache Foundation - Continuum
- Eclipse Foundation - Hudson
- Many in-house / script based systems

1.10.5 Continuous Integration and the Key Characteristics of Distributed Systems

- **Openness**
  - Several, including TFS and Team City, and any FOSS system allows for scripting, plugins, and expansion.
  - Some are more "canned" solutions like Team City, others are fully customizable like TFS
  - Most have a "free as in beer" mode that begins to cost money in wider deployments
  - Some of the FOSS alternatives can be a challenge to configure
- **Concurrency**
  - Continuous integration systems typically allow for replication and parallelism at the level of the build controller and in build nodes
  - When the cloud is used, resources consumption and cost can be flexible.
  - Supporting CI for 100s of developers can require a large amount of compute and I/O resources. These systems definitely have to scale.
- **Scalability**
  - Some systems are able to scale to the range of 10s of build nodes and controllers, most systems can scale to 100s
  - See http://tfs.visualstudio.com. This is a system that Microsoft has built that scales to the cloud level. You pay a subscription and use fee to use this.
- **Fault Tolerance**
  - Fault tolerance exists on several levels
  - **First is the build controller. Typically a primary / secondary replication system is employed.**
    - Issues of network partitioning aren't usually handled, just single-multi node failures
  - Second is the build node. The build controller will periodically poll build nodes to test their connectivity. If one node is lost, it is taken out of the pool. Any pending builds on such a system can be re-queued by the developer or automatically by the build controller
  - Source control and Artifact DB - these usually exist on systems with redundant and fast disk arrays that maintain daily backups. These systems can usually handle single hardware failures and continue to operate.
- **Transparency**
  - To the developer, the only thing that exists is the source control system and the build controller.
  - The build controller manages a queue of "builds" that a developer monitors.
  - **Whether there are 10s of build nodes or 100s, the developer is not aware of this detail or aware of where the buil**
    - transparency of location

    * transparency of scale

1.10.6 Case Study - My Simple CI System

1.10.7 The Front Page of Team City

- Build definitions for my active projects
- Several Windows and Linux software projects

Fig. 1.15: The Front Page of Team City

1.10.8 Build Nodes

- **3 Build Nodes**
  - Two Linux nodes
  - One Windows nodes
  - Supports the development of Windows and Linux software

1.11 Project Build Management With Maven

1.11.1 Introduction

- The goal of this page is to give you a step-by-step guide to get a Java project up and running with Maven 3.x
- The instructions are written against Maven 3.0.4 and Java 1.6.0.51 on Mac OSX 10.8.4

1.11.2 Creating a Maven Project from Scratch

- Maven has a concept of archetypes. Archetypes are skeleton projects that a user can use to get a basic development setup for one purpose or another (desktop application, servlet, etc. . .)
- The following command will invoke maven, fetch some initial dependencies, and list a set of archetypes that are available. If you press 'enter' at the first question, you will create a default Java desktop application. Maven will ask you to fill out some identifying information for your project (such as a project name, group name, etc. . .).

```
$ mvn archetype:generate
```

- **After this command finishes, you will have three interesting objects in your folder, a pom.xml, src/main, and src/test**
  - pom.xml contains the build, unit test, and dependency configuration
  - src/main contains the production code for your project
  - src/test contains the unit tests for the project.

1.11.3 Generating an Eclipse Project from a Maven Project

- After generating your Maven project, you won't be able to immediately open it in Eclipse
- Maven is able to generate the appropriate files to setup an eclipse project.
- This command will generate the necessary Eclipse project files

```
$ mvn eclipse:eclipse
```

1.11.4 Configuring Eclipse

- The created project should open in eclipse, but it is possible you might run into some problems.
- **Often, you may see an error related to M2_REPO not begin defined. To fix this**
    1. Right click on your project in the Package Explorer
    2. click on Properties in the context menu
    3. click on the Libraries tab
    4. click on the "Add Variable" button
    5. click "Configure Variables"
    6. click "New"
    7. Under name put M2_REPO
    8. Under Path, put your home folder and .m2/repository. On my system this is /Users/joe/.m2/repository
- Your project should build just fine. You are ready to develop your project.

1.11.5 Running Unit tests with Maven

- Running unit tests on the command line is simple with Maven, just run the following command

```
$ mvn test
```

- Maven will compile and run any unit test fixtures you have under src/test.

1.12 Writing with Sphinx

1.12.1 Introduction

Common document models such as Rich Text Format, or Microsoft Word formats are based around one properly encoded file or around a compressed archive containing several folders and files. In both cases, to the user, a single file is presented. This file will contain all text, formatting metadata, and non text objects such as pictures.

Alternatively, other document models like those used in LaTeX, Pandoc, Sphinx, and similar systems allow/require the author to create the individual parts of a document in separate files and typically with separate software packages. Theses systems consume these individual files to create a final document in one or more possible formats.

The first model often has the advantage of WYSIWYG (What You See Is What You Get) editors, and have all document data managed through one program that has one set of paradigms. This is a great way to go for many types of document authoring. The approach is simple to understand and the tools are relatively easy to use.

The second model often does not have a WYSIWYG editing experience. It, however allows you to use separate programs to manage each part of your document. For example, creating diagrams, authoring code, collecting tables of data, can all be done in a more comfortable way in non-text editor programs. This model also allows for a great deal of distributed authoring. When authoring books, it is possible for several contributors to work on the same document in parallel by working on separate files that contain different chapters of the document's text. Many of these tools also allow document authors to create several precisely formatted documents from one source of authored text and supplemental objects. For example, it is possible in Pandoc to create a paper formatted to the IEEE standard, a book chapter, and a webpage all by running the pandoc software with different arguments.

1.12.2 About Sphinx

Sphinx makes use of Restructured Text. A great explanation of reST can be found on WikiPedia here: http://en.wikipedia.org/wiki/ReStructuredText. Essentially restructured text allows the user to write text in plain ASCII using ASCII representations of common formatting such as underlines, bullets, numbered lists, and others. Sphinx can take this restructured text and translate it into a document that has the same kind of formatting but in a way that is native to that document format.

We note that Sphinx is not the only authoring framework of its kind. We use other tools based on a similar design for different situations. As examples:

- Jekyll is a framework based on Markdown (an alternative to reStructuredText) for static website generation. We use this for maintaining sites like http://thiruvathukal.com. While awesome in general, it is not the best solution when you want the ability to author documents intended for print (PDF) and e-reading (ePub). Sphinx does all of the above, so to speak.
- LaTeX itself could be used to author the content directly. LaTeX does a delightful job at its core competency, which is *typesetting*, which is clearly a relic of the print era. With a number of supplementatal tools, we can go to other formats. However, LaTeX has a rather steep learning curve for new users and is much more complex than reStructuredText and Markdown.
- Pandoc is a delightful tool that can target all of the different formats as Sphinx does. In fact, it generates some of the *cleanest* output we've seen from any tool. Unlike Sphinx, however, it lacks innate support for writing chapter-oriented works (a concept we still find useful) and also suffers from the drawback of making it virtually impossible to externalize code examples.

We think you'll find Sphinx as enjoyable as we do and might even wonder how it is possible to organize a meaningful book without support for its core ideas (especially books having code examples and mathematics!)

1.12.3 Additional Resources

1. http://sphinx-doc.org/rest.html
2. http://sphinx-doc.org/tutorial.html
3. http://pythonhosted.org/an_example_pypi_project/sphinx.html

1.12.4 Setting up Sphinx - Ubuntu Linux

Run the following commands to get a Sphinx environment setup on your Linux machine (as the super user):

1. apt-get update
2. apt-get install python python-setuptools python-pip make
3. easy_install -U sphinx
4. pip install sphinx_bootstrap_theme

If those commands all succeed, you will have sphinx installed. To test if the command is installed you can run "sphinx-build". This will launch the sphinx program. It will print its version and possible arguments.

1.12.5 Sphinx on non-Linux Platforms

If you are setting up Sphinx on Linux, you will by far have the easiest time getting things up and running. So, if that is an option for you, your best bet is to pursue it. Sphinx is supported on other platforms such as Windows and OSX, but the installation process isn't as reliable as it is on Linux and might take some more effort to get it running properly.

1.12.6 Setting up Sphinx - Mac OSX (Using HomeBrew instead of MacPorts)

Getting things up and running on OSX takes a bit more work. Here's essentially what needs to be done:

1. Install Mac HomeBrew - http://brew.sh
2. Using the brew command from HomeBrew, install python, python-setuptools and python-pip
3. easy_install -U sphinx
4. pip install sphinx_bootstrap_theme.

1.12.7 Setting up Sphinx - Windows

The Sphinx documentation is a great resource for installing on Windows. You can find it here - http://sphinx-doc.org/latest/install.html

1.12.8 Setting up Sphinx in a Python virtualenv

If you are working with a *copy* or *clone* of https://bitbucket.org/loyolachicagocs_books/distributedsystems, you might notice that the top level folder, contains a shell script, `sphinx.sh` that drives the Sphinx build process.

```
#! /bin/bash

[ -f ~/.env/sphinx/bin/activate ] && . ~/.env/sphinx/bin/activate

rm -rf build/
make html
make epub
make LATEXOPTS=' -interaction=batchmode ' latexpdf
```

In this example, there is the following line that checks (my setup) for the presence of a Python virtual environment.

```
[ -f ~/.env/sphinx/bin/activate ] && . ~/.env/sphinx/bin/activate
```

What this achieves is to give you your own Python interpreter and set of libraries and Python tools if the virtualenv can be successfully activated.

On George's system, observe what the following does:

```
$ . ~/.env/sphinx/bin/activate
(sphinx)atomium:distributedsystems gkt$ which python
/Users/gkt/.env/sphinx/bin/python
(sphinx)atomium:distributedsystems gkt$ which sphinx-build
/Users/gkt/.env/sphinx/bin/sphinx-build
```

First, ensure the `virtualenv` command is present:

```
$ pip install virtualenv
```

Create the virtualenv. It can be anywhere you like, but I keep all of my virtualenv instances in the ~/.env directory (.env in my home folder). You can put it anywhere you like:

```
$ virtualenv ~/.env/sphinx
```

This generates a bunch of output:

```
$ virtualenv ~/.env/sphinx
New python executable in /Users/gkt/.env/sphinx/bin/python2.7
Also creating executable in /Users/gkt/.env/sphinx/bin/python
Installing Setuptools.....................................................................
Installing Pip...........................................................................
```

Activate the virtualenv:

```
$ . ~/.env/sphinx/bin/activate
(sphinx)imac-g5:~ gkt$
```

Install the `sphinx` and `sphinx_bootstrap_theme`. Both of these are needed to rebuild the course notes.

```
$ pip install sphinx sphinx_bootstrap_theme

(you'll get a LOT of output, not shown here but ending with the following)

Successfully installed sphinx sphinx-bootstrap-theme Pygments Jinja2 docutils markupsafe
Cleaning up...
```

You are unlikely to have any problems if you do this on OS X or Windows. If you are on Linux, it is likely that you need some additional packages (already covered in our Ubuntu section above).

virtualenv has the added advantage of allowing you to do Python experimentation with as little use of the *root* user as possible. Once you have it going, you'll come to realize that you can't live without it, especially when doing Python work!

1.12.9 Authoring in Sphinx

An excellent way to learn Sphinx is to look at existing documents. The concurrency lecture has several example usages of reST: http://distributed.cs.luc.edu/html/_sources/concurrency.txt. The entire source repository for the distributed systems lecture notes is available on BitBucket here - https://bitbucket.org/loyolachicagocs_books/distributedsystems. You can download the source, and see an entire working example.

You are encouraged to look through the Sphinx documentation for full details.

1.12.10 Examples

Creating a Heading:

```
This Is a Heading Underlined With Dashes To Make It A Heading
-------------------------------------------------------------
```

Creating a Heading with bullets under it

```
My List (notice indentation and space after dash)
-------------------------------------------------
        - Item 1
        - Item 2
```

Creating a heading with a numbered list under it (numbers are generated by sphinx automatically)

```
My Numbered List (notice space after the pound and dot)
-------------------------------------------------------
        #. First Item
        #. Second Item
        #. Third Item
```

Including a figure from a PNG image

```
.. figure:: figures/figure01.png
        :align: center
        :width: 600px
        :alt: Figure 01
```

Creating an inline code snippet (Approach 1) (notice the blank line after the two colons and the indentation)

```
::
        #include <stdio.h>

        int main() {
                printf("Hello World\n);
                return 0;
        }
```

Creating an inline code snippet (Approach 2) (this approach uses an actual code file, it will start the snippet from the line in the file matching the text after the "start-after" and end the snippet before the "end-before". Your best bet is to add the pattern .. in this case begin-main-function and end-main-function as comments in your example code file.)

```
.. literalinclude:: examples/main.c
        :start-after: begin-main-function
        :end-before: end-main-function
        :linenos:
```

1.13 Clocks and Synchronization

1.13.1 What is Time? Some Physics

- The understanding of time has evolved greatly in the past 150 years.
- **With Einstein's Special and General Relativity Theories, we've come to understand:**
  - That time is interwoven with space
  - Time is relative to the reference frame of an observer
  - We now know that simultaneity is a relative concept. Two events A and B occurring simultaneously in one reference frame may appear to be ordered A then B in another frame and B then A in yet another reference frame.
  - All of this is true without invalidating the observations our causal relationships observed in any reference frame.

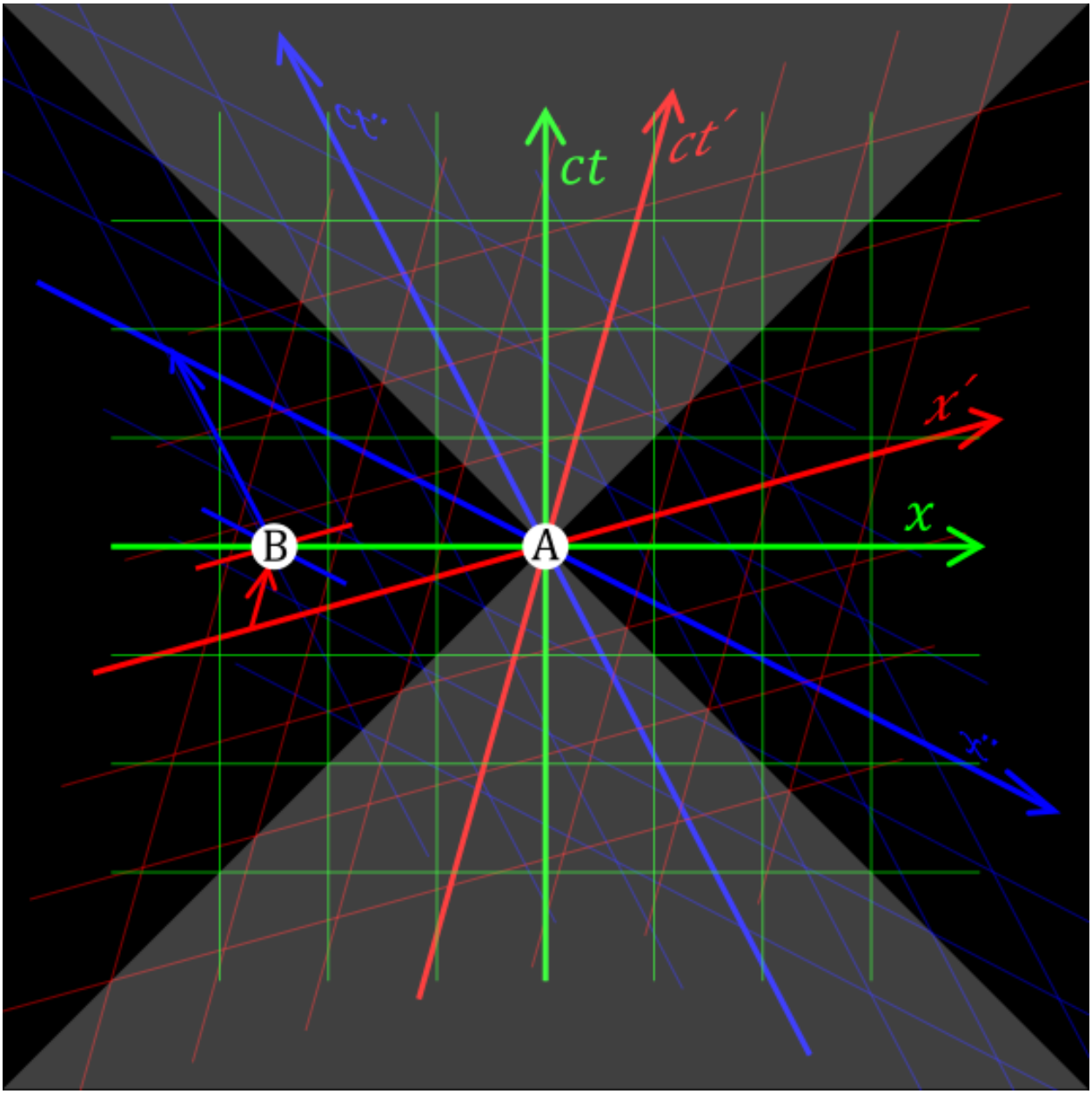


Fig. 1.16: Two events in space-time. The green observer sees A and B happening at the same time since the two events happen on the same X time plane for A. For the red observer, B is encountered first, then A second. For the blue observer A happens first, and B second. (Image taken from http://commons.wikipedia.org/wiki/File:Relativity_of_Simultaneity.* under the Creative Commons License.)

1.13.2 Time and Computation

- This dive into physics is not so much to give a lesson in physics, but to impress upon you the underlying complexity of something most people take for granted every day.
- In a similar fashion, in computation, the concept of time measurement can be very complicated
- When building a distributed system, do not take it for granted that you can simply trust the clock on the machine that is executing your code.
- More complicated solutions are needed to establish the order of events that have occurred or when they will occur in the future.

1.13.3 Physical Clocks

- Computer Timer: an integrated circuit that contains a precisely machined quartz crystal. When kept under tension the quartz crystal oscillates at a well-defined frequency.
- Clock Tick: after a predefined number of oscillations, the timer will generate a clock tick. This clock tick generates a hardware interrupt that causes the computer's operating system to enter a special routine in which it can update the software clock and run the process scheduler.
- **This system is fairly reliable on one system. With the timer we can define:**
  - simultaneous: all actions that happen between clock ticks
  - before: an operation that happens in a previous clock tick
  - after: an operation that happens in a subsequent clock tick

1.13.4 Physical Clocks - Multiple Systems

- Unfortunately, it is impossible for each machined quartz crystal in every computer timer to be exactly the same. These differences create clock skew.
- For example, if a timer interrupts 60 times per second, it should generate 216,000 ticks per hour.
- In practice, the real number of ticks is typically between 215,998 and 216,002 per hour. This means that we aren't actually getting precisely 60 ticks per second.
- We can say that a timer is within specification if there is some constant p such that:

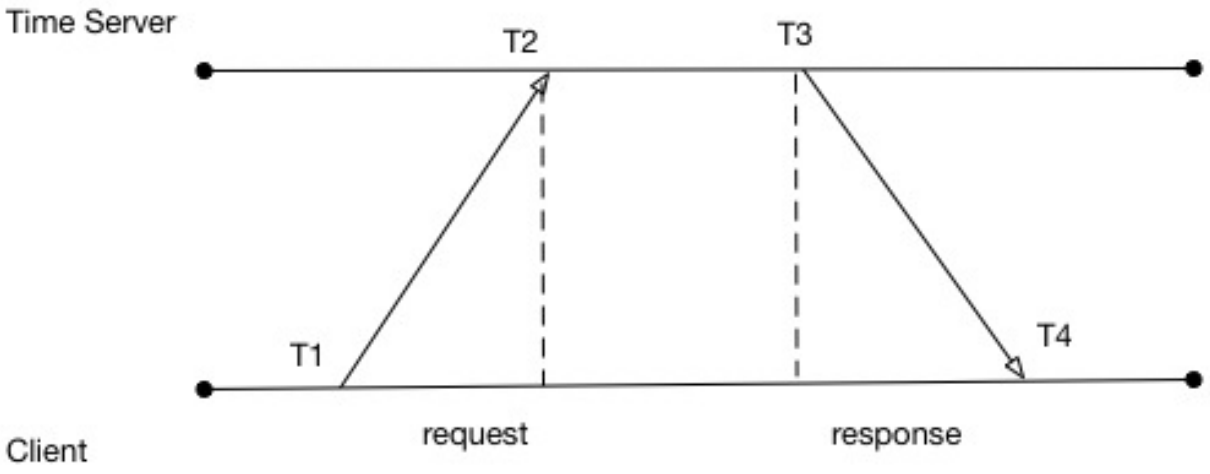
$$1 - p <= \frac{dC}{dT} <= 1 + p$$

- The constant p is the maximum drift rate of the timer.
- On any two given computers, the drift rate will likely differ.
- To solve this problem, clock synchronization algorithms are necessary.

1.13.5 Clock Synchronization

- The common approach to time synchronization has been to have many computers make use of a time server.
- Typically the time server is equipped with special hardware that provides a more accurate time than does a cheaper computer timer

- The challenge with this approach is that there is a delay in the transmission from the time server to the client receiving the time update.
- This delay is not constant for all requests. Some request may be faster and others slower.
- So how do we solve this problem?



- The relative time correction C can be calculated as:

$$C = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

- The way this works is that the client sends a packet with T_1 recorded to the time server. The time server will record the receipt time of the packet T_2. When the response is sent, the time server will write its current time T_3 to the response. When the client receives the response packet, it will record T_4 from its local clock.
- When the value of C is worked out, the client can correct its local clock
- The client must be careful. If the value of C is positive, then C can be added to the software clock
- If the value of C is negative, then the client must artificially decrease the amount of milliseconds added to its software clock each tick until the offset is cleared.
- It is always inadvisable to cause the clock to go backwards. Most software that relies on time will not react well to this.

1.13.6 Clocks Synchronization and Reliability

- Always remember: the goal of clock synchronization is to minimize the difference between the accepted actual time and the time on a given client machine
- When this is achieved, it can be said that in a given set of computers that synchronize that their clocks are more closely in sync.
- Even in this case of more accurate and more often corrected for clocks, developers of distributed systems should still be wary of relying on local clock time.
- Because there are corrections going on, the time recorded for an event might have actually happened at a different time on another computer because of differing drift rates of those computer timers.
- Because we have correction of time does not mean that all machines agree on time, it just means they are much closer to each other on average.

- For some distributed systems, this may be sufficient, for others it may not be.

1.13.7 Lamport's Logical Clocks

- An important paper to read - "Time, clocks, and the ordering of events in a distributed system" by Lamport (1978).
- This paper can be looked up on scholar.google.com
- The important contribution of Lamport is that in a distributed system, clocks need not be synchronized absolutely.
- If two processes do not interact, it is not necessary that their clocks be synchronized because the lack of synchronization would not be observable and thus not cause problems.
- It is not important that all processes agree on what the actual time is, but that they agree on the order in which events occur.
- **Rules of Lamport's Logical Clocks:**
  - Defines a relationship called "happens-before". a -> b is read as "a happens before b"
  - if a and b are events in the same process and a occurs before b, then a -> b is true.
  - if a is the event of a message being sent by one process and b is the event of the message being received by another process, then a -> b is true
  - "happens-before" is transitive, meaning if a -> b and b -> c, then a -> c
  - **if a -> b happens between two process, and events x and y occur on another set of processes and these two sets of**
    - we cannot say whether x -> y or y -> x from the perspective of the first set of processes

1.13.8 Implementing Lamport's Logical Clocks

- When a message is transmitted from P1 to P2, P1 will encode the send time into the message.
- When P2 receives the message, it will record the time of receipt
- If P2 discovers that the time of receipt is before the send time, P2 will update its software clock to be one greater than the send time (1 milli second at least)
- If the time at P2 is already greater than the send time, then no action is required for P2
- With these actions the "happens-before" relationship of the message being sent and received is preserved.

1.13.9 Limitations of Lamport's Logical Clocks

- Lamport's logical clocks lead to a situation where all events in a distributed system are totally ordered. That is, if a -> b, then we can say $C(a) < C(b)$.
- Unfortunately, with Lamport's clocks, nothing can be said about the actual time of a and b. If the logical clock says a -> b, that does not mean in reality that a actually happened before b in terms of real time.
- The problem with Lamport clocks is that they do not capture causality.
- If we know that a -> c and b -> c we cannot say which action initiated c.
- This kind of information can be important when trying to replay events in a distributed system (such as when trying to recover after a crash).

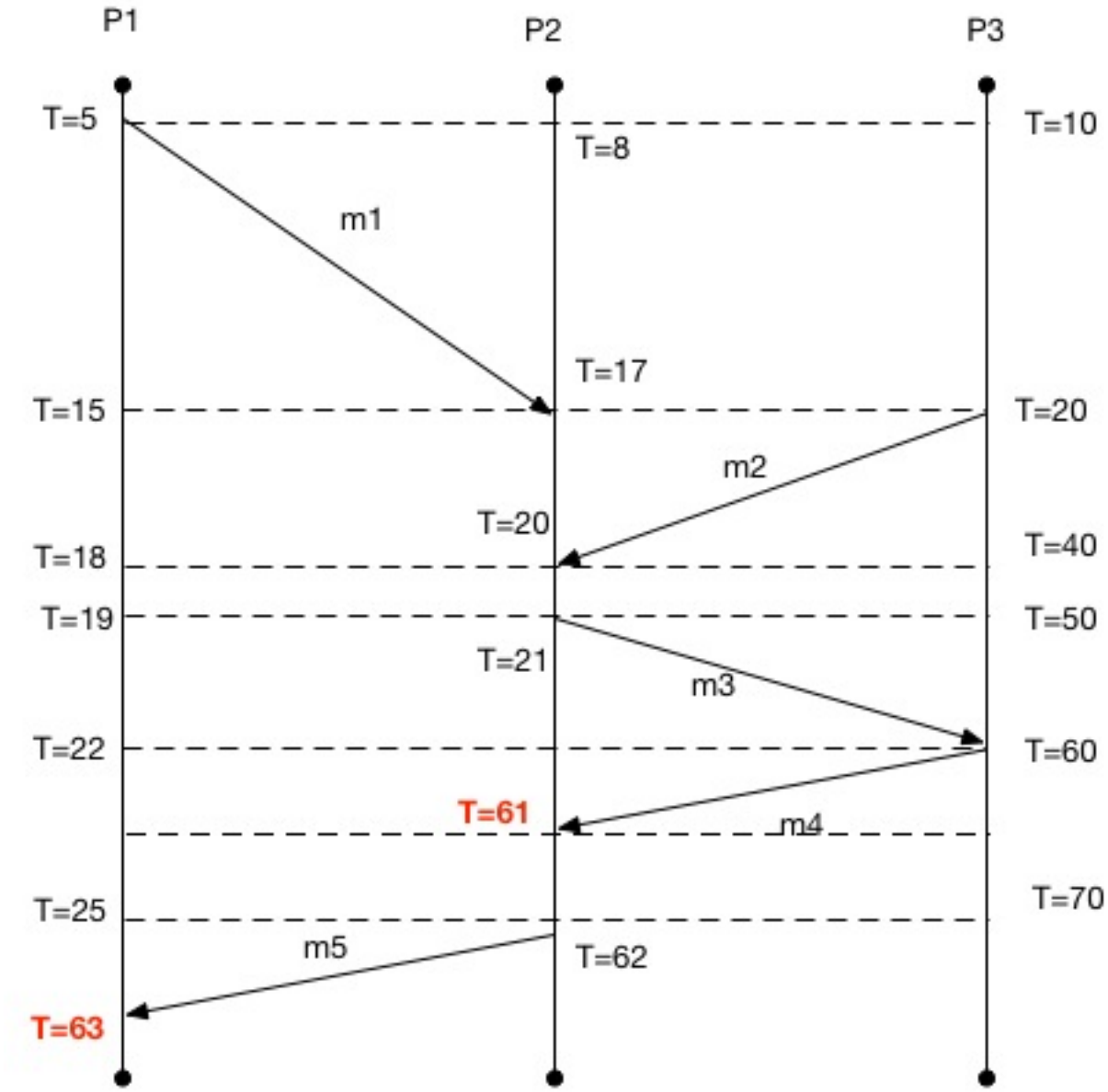


Fig. 1.17: From this diagram, we can see that $m_1 -> m_3$. We also know that $C(m_1) < C(m_3)$. We can see that $m_2 -> m_3$ and that $C(m_2) < C(m_3)$. What we cannot tell here is whether m_1 or m_2 caused m_3 to be sent.

- The theory goes that if one node goes down, if we know the causal relationships between messages, then we can replay those messages and respect the causal relationship to get that node back up to the state it needs to be in.

1.13.10 Vector Clocks

- Vector clocks allow causality to be captured
- **Rules of Vector Clocks:**
  - A vector clock $VC(a)$ is assigned to an event a
  - If $VC(a) < VC(b)$ for events a and b, then event a is known to causally precede b.
- **Each Process P_i maintains a vector VC_i with the following properties:**
  - $VC_i[i]$ is the number of events that have occurred so far at P_i. i.e. $VC_i[i]$ is the local logical clock at process P_i
  - If $VC_i[j] = k$ then P_i knows that k events have occurred at P_j. It is thus P_i's knowledge of the local time at P_j

1.13.11 Implementing Vector Clocks

- The first property of the vector clock is accomplished by incrementing $VC_i[i]$ at each new event that happens at process P_i
- **The second property is accomplished by the following steps:**
  1. Before executing any event (sending a message or some internal event), P_i executes $VC_i[i]$ <- $VC_i[i] + 1$
  2. When process P_i sends a message m to P_j, it sets m's (vector) timestamp $ts(m) = VC_i$
  3. Upon receiving a message m, process P_j adjusts its own vector by setting $VC_j[k]$ <- $max(VC_j[k], ts(m)[k])$ for each k.

1.13.12 So... What Did We Get Out of All of This?

- We can say if an event a has a timestamp $ts(a)$, then $ts(a)[i] - 1$ denotes the number of events processed at P_i that causally precede a
- This means that when P_j receives a message from P_i with timestamp $ts(m)$, it knows about the number of events that occured at P_i that causally preceded the sending of m
- Even more importantly, P_j has been told how many events in **other** processes have taken place before P_i sent message m.
- So, this means we could achieve a very important capability in a distributed system: we can ensure that a message is delivered only if all messages that causally precede it have also been received as well.
- We can use this capability to build a truly distributed dataflow graph with dependencies without having a centralized coordinating process.

1.14 Domain Name Service

1.15 Message Passing Interface

1.16 Object Brokers and CORBA

1.17 REST and Web Services

1.18 NoSQL and Sharding

CHAPTER

TWO

LEGACY CONTENT

2.1 Legacy Lectures

2.1.1 Software

- Languages
- Distributed Version Control Tools

2.1.2 Introduction

- Issues in Distributed Systems

2.1.3 Networking Overview

- Networking Primer and Protocols
- Sockets Programming Examples from HPJPC

2.1.4 Threads and Processes

- Threads Primer
- Examples from HPJPC

2.1.5 Filesystems

- Distributed Filesystems (NFS, AFS, etc.)
- Userland Filesystems

2.1.6 Filesystems Research at LUC

- NOFS and RESTFS

2.1.7 Grids and Clouds

2.1.8 Message Passing / Messaging Middleware

- MPI
- 0mq

2.1.9 Distributed Object Programming

- CORBA Tutorial

2.1.10 Modern Distributed Databases

- MongoDB

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search