Summary Distributed Systems HS 2009

Introduction

- In the early times, computers were standalone devices
- Today, networking has become a fundamental part of computers
- Definition of a distributed system:
  - "A collection of independent computers that appears to its users as a single coherent system."
  - Hardware: the machines are autonomous
  - Software: the users think they deal with a single system

Challenges in Distributed Systems

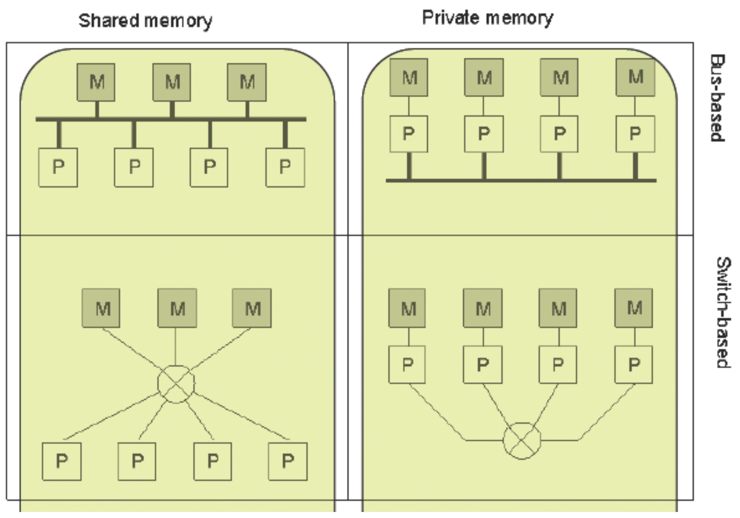
- Transparency
  - Make a set of computers appear as a single computer to the applications
  - Different types of transparency:
    - **Access**: hide differences in data representation and how a resource is accessed

      Problems: e.g. little endian representation vs. big endian representation or case-sensitive file system (Linux) vs. not case-sensitive file system (Windows)
    - **Location:** hide where a resource is located
    - **Migration:** hide that a resource may be moved to another location
    - **Relocation:** hide that a resource may be moved to another location while in use
    - **Replication:** hide that there may exists multiple replicas of a given resource
    - **Concurrency:** hide that a resource may be shared by several competitive users
    - **Failure:** hide the failure and recovery of a resource
    - **Persistence:** hide whether a (software) resource is in memory or on disk
- Heterogeneity
  - It is necessary to transparently address the differences in performance, capabilities, network connectivity, etc.
  - For instance in a PAN, connecting a desktop, a laptop, a PDA and a mobile phone, one should transparently avoid to assign intensive tasks to the mobile phone.
- Failure handling
  - Distributed Systems should be failure transparent → a failure on some components, should not be fatal (or, ideally even detectable to the applications)
- Openness
  - Well defined interfaces should be used. The interfaces should be described using the Interface Definition Language (IDL)
  - Modularity should be used so that an upgrade of one module doesn't affect the rest
- Scalability
  - 3 different dimensions of scalability: Size scalability, Geographic scalability and Administrative scalability

    - Some factors can limit the scalability (e.g. centralized services, centralized data or centralized algorithms (doing routing base on complete information)).
    - Techniques to fight scalability problems
        - Distribution of responsibilities (e.g. DNS name space)
        - Hide communication latencies (e.g. client check forms as they are being filled instead of the server checks the form)
        - Apply replication techniques (e.g. caching)
- Security
    - Security is the weakest link in Distributed Systems

Hardware Architectures

- 2 different hardware concepts:
    - Multiprocessors: multiple processors share a pool of memory
    - Multicomputers: multiple processor with private memory are interconnected

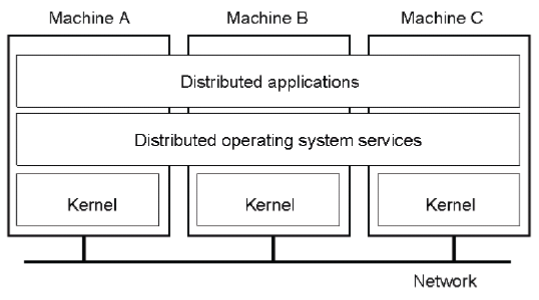


Hardware concept	PLUS	MINUS
Multiprocessors, bus-based	Simpler and cheaper construction	Not scalable: with more than a few processors, the bus is saturated
Multiprocessors, switch-based	Increased concurrency → gives speed	Delay due to many switches, expensive linkage & fast crosspoint switches

Software Architectures

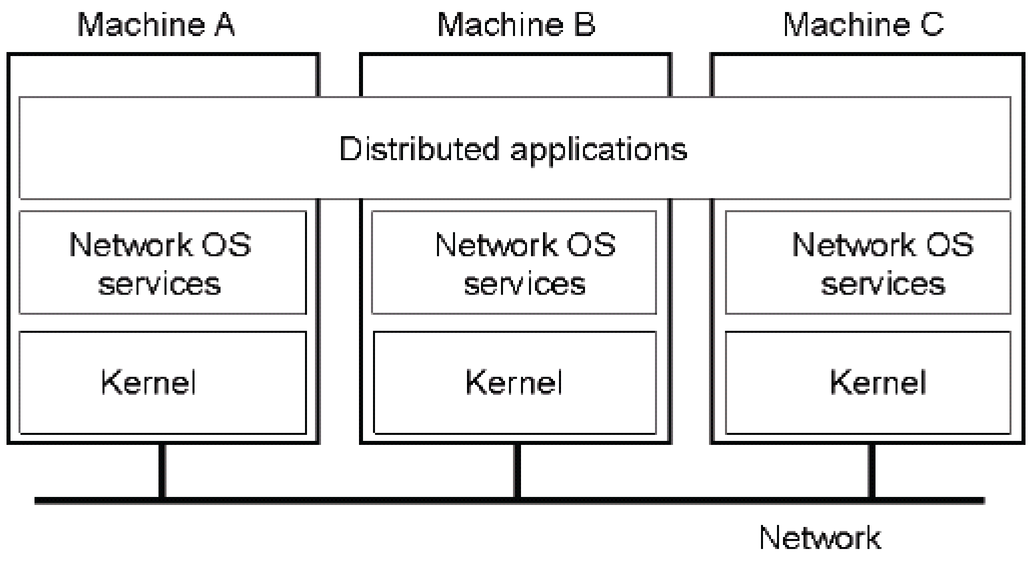
System	Description	Main Goal
Distributed OS	Tightly-coupled operation system for multiprocessors and homogeneous multicomputers	Hide and manage hardware resources
Network OS	Loosely-coupled operating system for heterogeneous multicomputers (LAND and WAN)	Offer local services to remote clients
Middleware	Additional layer atop of NOS implementing general purpose services	Provide distribution transparency

Distributed OS



- Takes care of:
    - Transparent task allocation to a processor
    - Transparent memory access (Distributed Shared Memory)
    - Transparent storage
- Provides complete transparency and single view of the system
- Requires multiprocessors or homogenous multicomputers

Network OS



- Provides services (e.g. ftp, nfs, rlogin)
- Not transparent, no single view of the system
- Very flexible with respect to heterogeneity and participation
- Problem: Different clients may mount the servers in different places

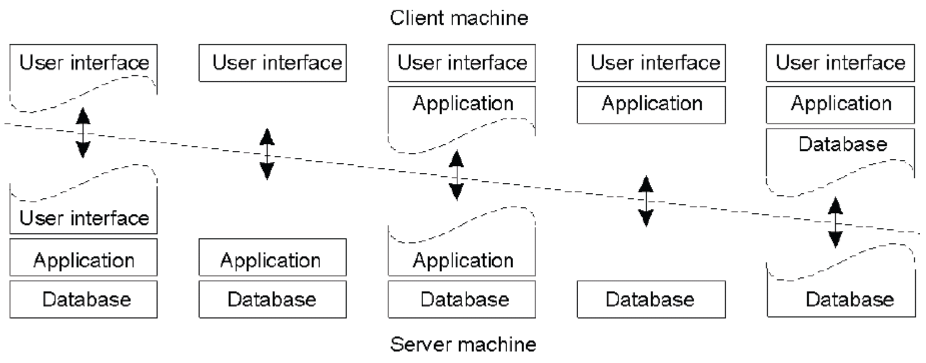
Comparison between Systems

Item	**Distributed OS**		**Network OS**	**Middleware-based OS**
	Multiproc.	Multicomp.		
Degree of transparency	Very high	High	Low	High
Same OS on all nodes	Yes	Yes	No	No
Number of copies of OS	1	N	N	N
Basis for communication	Shared memory	Messages	Files	Model specific
Resource management	Global, central	Global, distributed	Per node	Per node

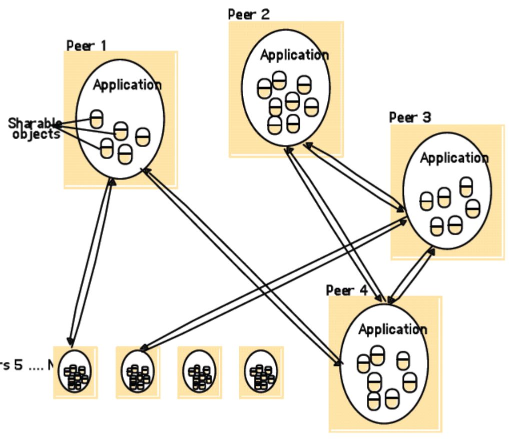
Scalability	No	Moderately	Yes	Varies
Openness	Closed	Closed	Open	Open

Types of Network Interaction

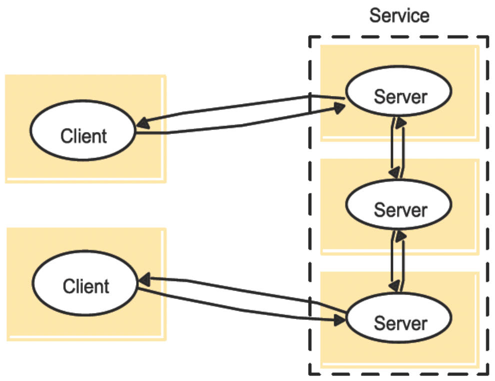
- Client/Server: synchronous call (Client waits for the result)
- 3-tier network application: user-interface level, processing level and data level
- Multi-tiered architecture:



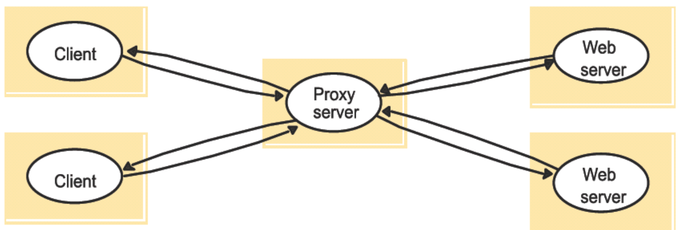
- Peer processes:



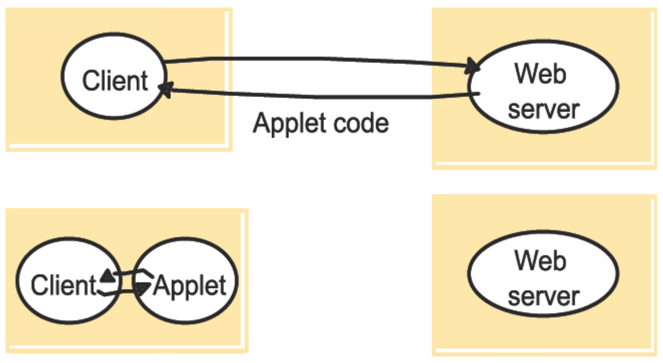
- Cluster of servers:



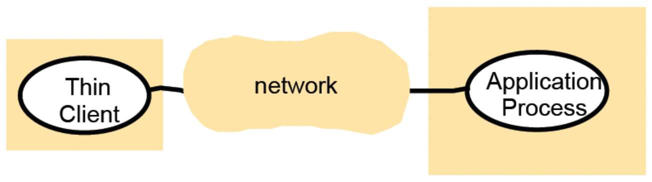
- Web proxy server:



- Code mobility:



- Thin clients:

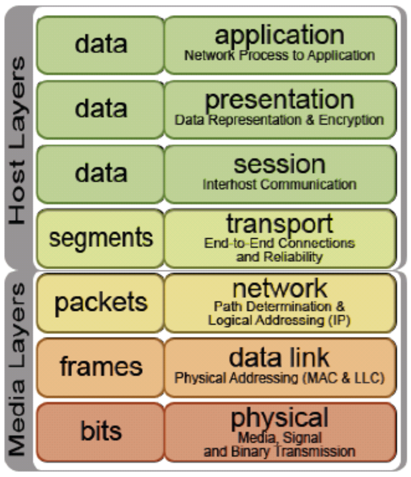


Inter-Process Communication

Communication layers

- Processes on different computers need to exchange information
- It is necessary to abstract: concentrate on what data to exchange and with whom and ignore how that data is transferred
- Communication takes place by exchanging messages → many agreements are needed at many different levels

ISO/OSI Layers


Host Layers
Media Layers
data
application
Network Process to Application
data
presentation
Data Representation & Encryption
data
session
Interhost Communication
segments
transport
End-to-End Connections and Reliability
packets
network
Path Determination & Logical Addressing (IP)
frames
data link
Physical Addressing (MAC & LLC)
bits
physical
Media, Signal and Binary Transmission

Layers 1 to 3 are used to interact between consecutive nodes in the internet infrastructure (bridges, routers)

Layers 4 to 7 are used for end-to-end interaction

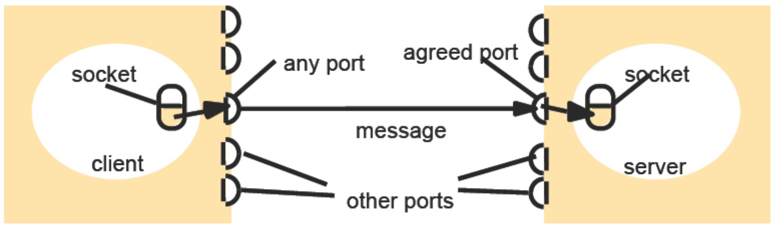
- **Layer 1:** electrical/mechanical/optical signaling interfaces
- **Layer 2:** Groups bits into frames and adds some extra information (starting and ending bit patterns, sequence number, checksum)
- **Layer 3:** routes packets towards the destination (most common protocol: IP (Internet Protocol))

- **Layer 4:**
    - provides end-to-end functionality
    - Most known protocols: UDP and TCP
    - Splits applications messages into packets (message fragmentation)
    - Reliable and in-order delivery
- **Layer 5:** used for synchronization, but not used in practice
- **Layer 6:** deals with the meaning of bits
- **Layer 7:** all distributed systems are here (protocols like FTP, HTTP, SSH, SMTP, …)

Type of connections

- Connection-oriented
    - Before communication, sender & receiver negotiate what protocols and parameters will be used
    - When done, terminate the connection
    - The sender sends a stream of bytes that transparently get grouped in packets and delivered to the receiver
    - Analogous to making a phone call
- Connectionless
    - No setup & no termination
    - The sender explicitly sends individual packets to the receiver
    - Analogous to sending letters by post

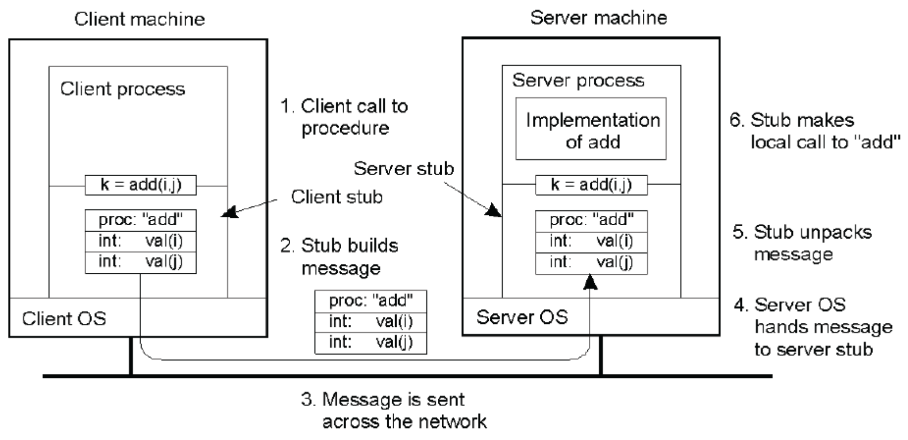
Working with Sockets



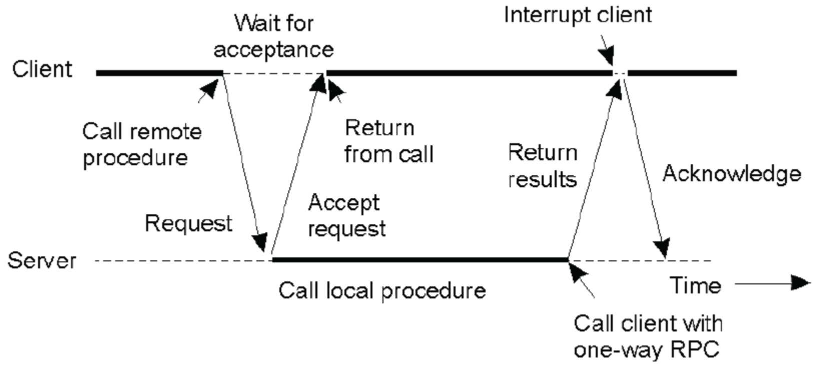
- UDP and Sockets: → multiple clients can be accessing the server intermittently
- TCP and Sockets: → we need threads to support multiple concurrent clients

Remote Procedure Calls

- Remote procedure calls constitute a middleware-layer functionality between layer 6 (presentation) and layer 7 (application)

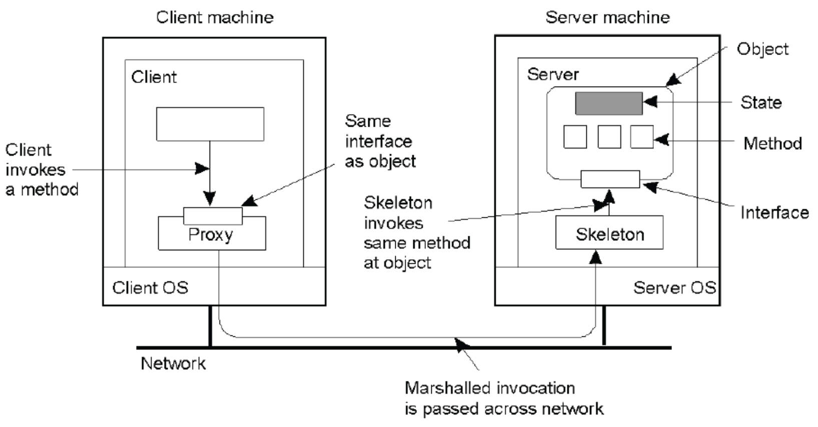


- Problems with RPC
    - Data representation (different encodings)
    - Passing arguments (pass-by-reference)
- Passing arguments
    - Arguments passed by reference are passed by copy/restore
- Asynchronous RPC



Distributed Objects

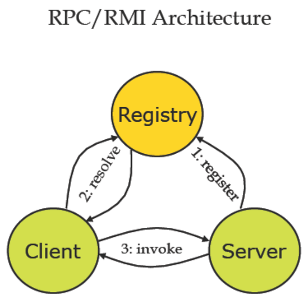
- Common organization using proxy object:

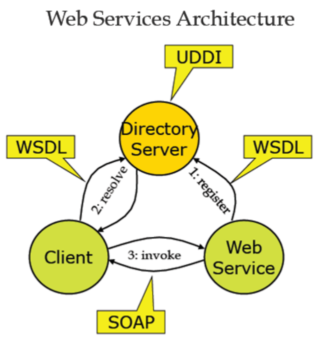


- The Distributed-Object-Model in DCE (Distributed Computing Environment)
    - Distributed dynamic (private) objects
    - Distributed named (shared) objects

Web Services

Architecture Overview

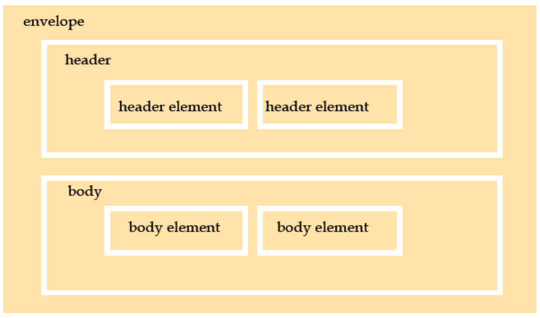


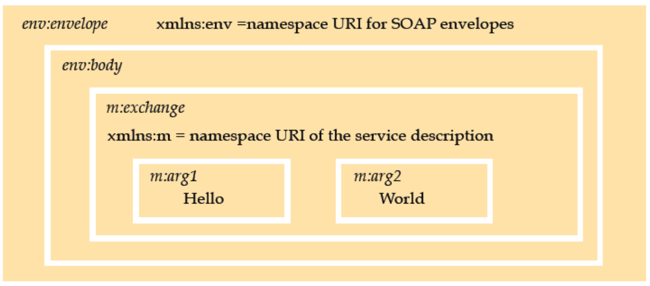


- Basic parts
    - Wire protocol
        - Used for the interaction between remote sites
        - Should work over any transport protocol (therefore it should be based on messages instead of procedure calling)
    - Description of Web Services
        - WSDL (Web Service Description Language): standardized and XML-based way to describe service interfaces
    - Discovery of Web Services
        - UDDI (Universal Description, Discovery and Integration) is used like a registry

SOAP

- Provides interoperability at the lowest level
- Defines a message format encoded in XML
- Defines how a client can invoke a remote procedure by sending a SOAP message, and how the server can reply by sending another SOAP message back

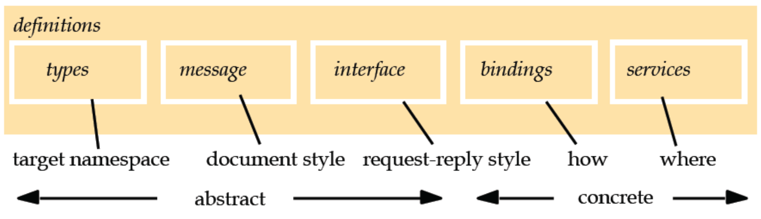



env:envelope
xmlns:env =namespace URI for SOAP envelopes
env:body
m:exchange
xmlns:m = namespace URI of the service description
m:arg1
Hello
m:arg2
World

Each box represents an XML element with its name in italics followed by any attributes and its content

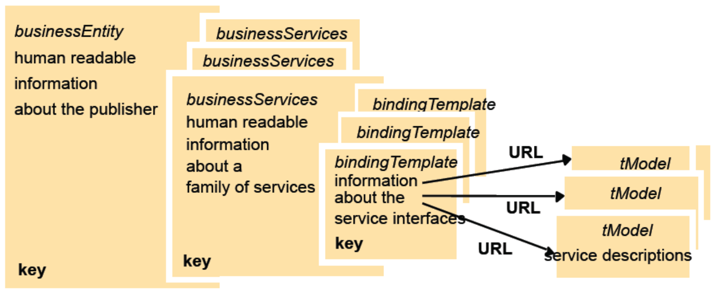
WSDL

- WSDL = Web Service Description Language
- XML syntax for formally describing how to invoke a web service and to communicate with it



UDDI

- Provide a standard, flexible way to discover where a web service is located and where to find more information about what the web service does
- Provides a registry function for managing information about web services
- Data structure:



- **businessEntity:** die Komponente beschreibt die Organisation, die den Web Service anbietet hinsichtlich allgemeine Identifikations- und Kontaktangaben
- **businessService:** hier sind die jeweils angebotenen Web Services charakterisiert; dabei kann es sich um mehrere Service-Angebote aber auch um mehrere technologische Formen ein- und desselben Web Service handeln
- **bindingTemplate:** diese Komponente gibt nun die detaillierten technischen Informationen zur Nutzung des Web Service an; dabei wird auf jeweilige Service-Beschreibungen als technical Model (tModel) referenziert
- **tModel:** bei dieser Komponente handelt es sich um einen generischen Container, der die detaillierten Service-Informationen zusammenfasst
- Page Model
    - **White Pages**: für die Kontaktinformationen zum Web-Service-Anbieter und einer allgemeinen Service-Charakterisierung
    - **Yellow Pages**: als kategorisierte Beschreibung des jeweiligen Web Services
    - **Green Pages**: für die detaillierten (technischen) Angaben zur Nutzung des Web Services.

Web Services vs. CORBA

- CORBA is designed to run in an organization, Web Services are designed to run on Internet
- In CORBA type identifiers refer to ORB-repository and aren't generally understood
- HTTP/XML are simple, CORBA has a learning curve
- XML isn't as efficient as CORBA with its binary formats
- CORBA has transactions, concurrency control, security, access control and persistent objects

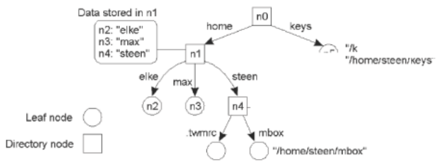
Naming and Location

Introduction (Why do we need naming?)

- Naming provides an abstraction useful for
    - Providing location independence
    - Allowing the relocation of entities
    - Allowing a single reference to a set of alternative access points
    - In some cases, for offering human-friendly names
- 3 general categories of naming types
    - Hierarchical naming
    - Flat naming
    - Attribute-based naming

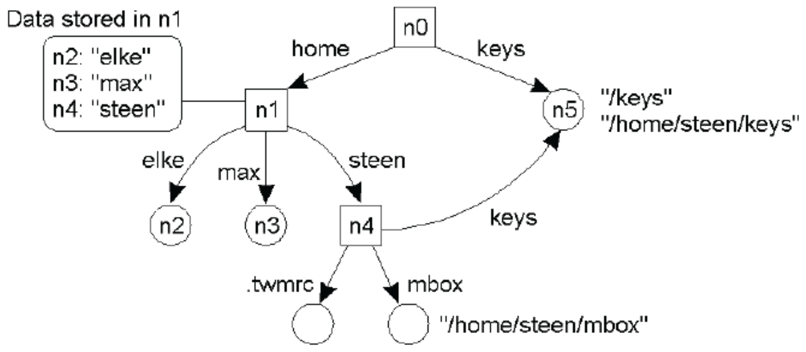
Hierarchical naming

- Based on the concepts of name spaces (a graph of names)
- Each name is a path in the naming graph (absolute if starting from the root, relative otherwise)

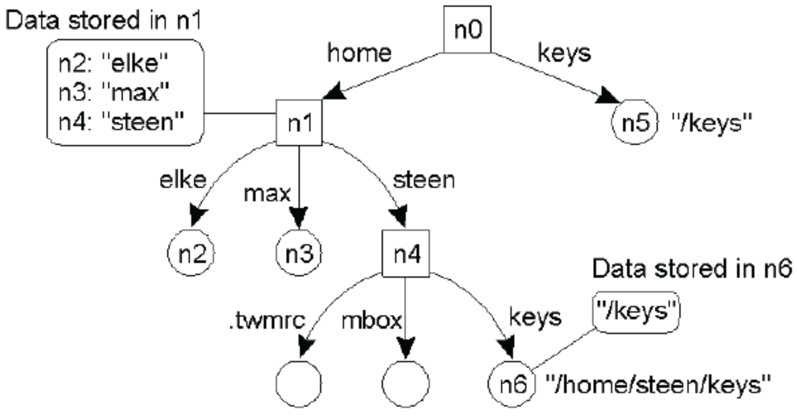


A general naming graph with a single root node.

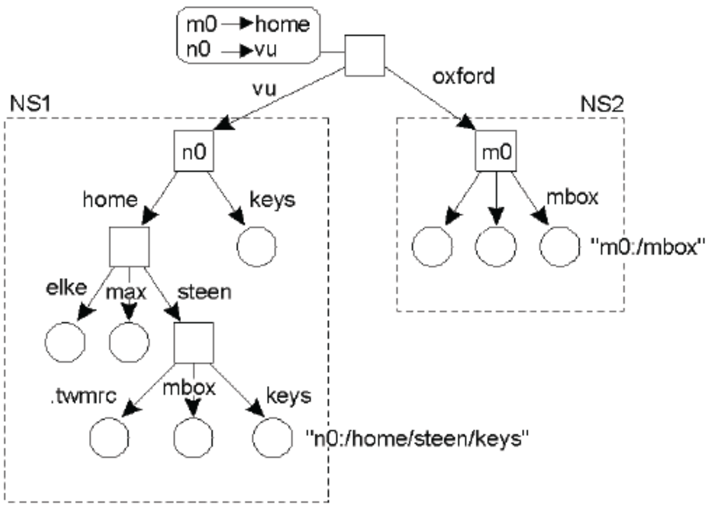
- Hard Links
  - An entity may have multiple names within a namespace → multiple paths that lead to the same leaf node



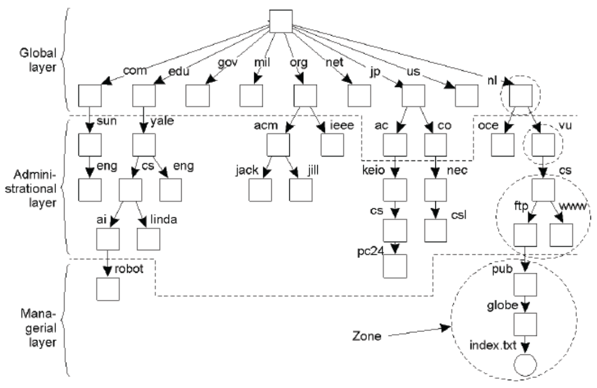
- Symbolic Links
  - A special node contains the absolute (or relative) name of another node



- Mounting:
  - A symbolic link may be referring to a remote name space, through a specific process protocol
- Merging name spaces
  - Adding a new root and mounting two or more namespaces below it
  - Problem: absolute names of all namespaces are changed
  - Solution: at root node cache the original top-level names
    e.g. the root remembers that home, keys map to /vu and mbox maps to /oxford



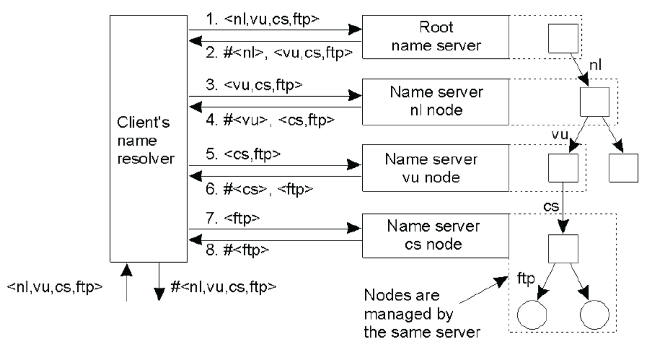
- DNS
    - A distributed directory service
    - Hierarchical name space (each level separated by '.')
    - One global root
    - Because of caching, queries to root servers are relatively rare
    - 3 major components
        - Domain name space and resource records
            - Specification of a tree-structured name space and data associated with names
        - Name servers
            - Hold information about a name space subset (zone) and have pointers to other name servers
            - May be authority for a zone (have full information about it)
        - Resolvers
            - Client programs that extract information from name servers

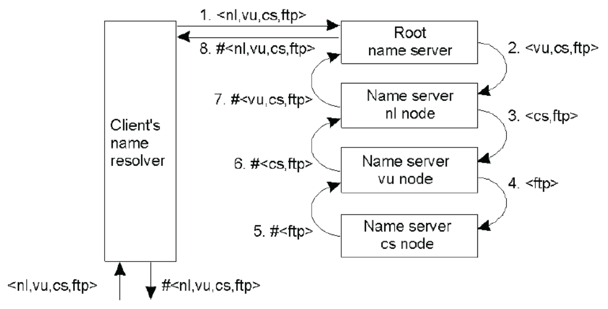


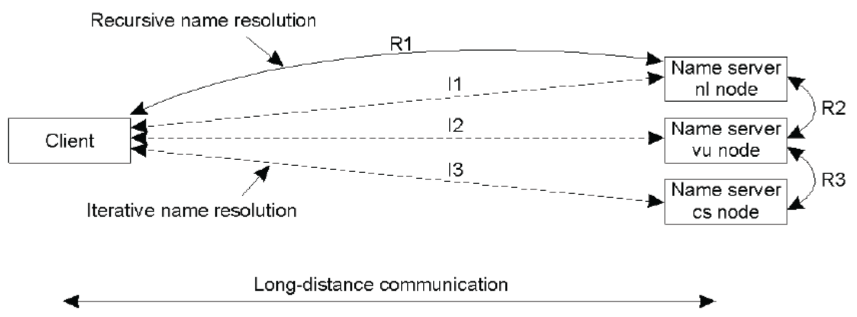
- DNS layers

Item	Global	Administrational	Managerial
Geographical scale of network	Worldwide	Organization	Department
Total number of nodes	Few	Many	Vast numbers
Responsiveness to lookups	Seconds	Milliseconds	Immediate
Update propagation	Lazy	Immediate	Immediate
Number of replicas	Many	None or few	None
Is client-side caching applied?	Yes	Yes	Sometimes

- Iterative and recursive name resolution





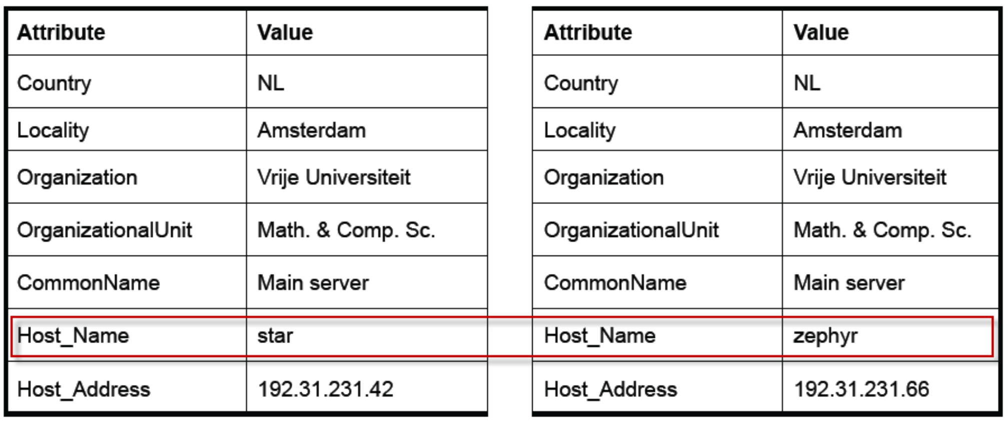

Recursive name resolution
R1
I1
I2
I3
R2
R3
Client
Name server nl node
Name server vu node
Name server cs node
Iterative name resolution
Long-distance communication

Flat Naming

- Useful when we want to address a space in a homogeneous way (e.g. memory addressing)
- Very common in centralized systems → in decentralized systems very complicated
- Naïve approach to resolve a name
    - Flood all networks asking who has the name in question
    - The node that has that name replies
    - Problem: doesn't scale well
    - Solution: Distributed Hash Tables

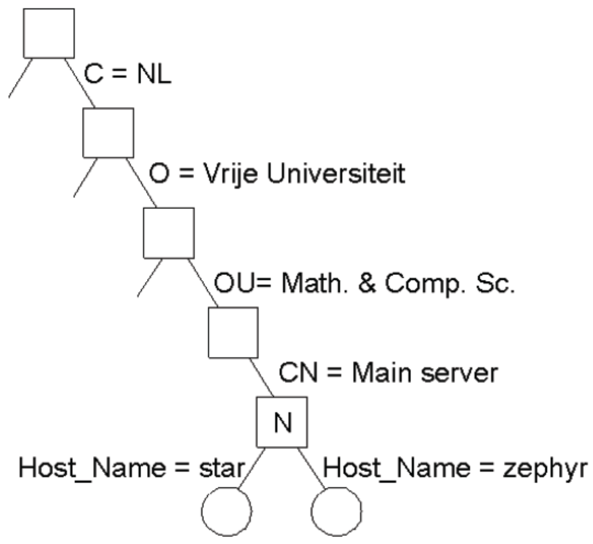
Attribute-based Naming

- Two X.500 directory entries having Host_Name as RDN (relative distinguished name)



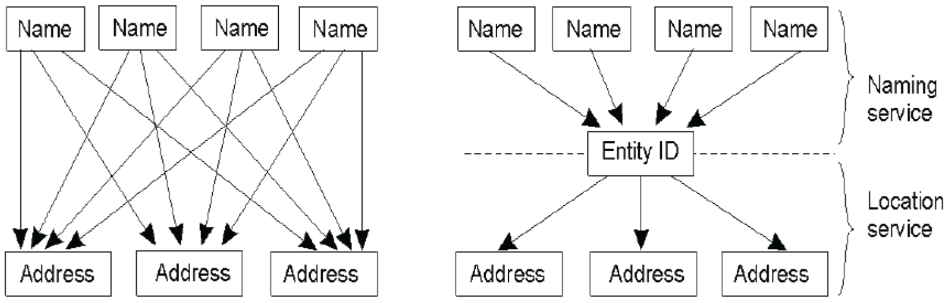
Attribute	Value
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Math. & Comp. Sc.
CommonName	Main server
Host_Name	star
Host_Address	192.31.231.42

Attribute	Value
Country	NL
Locality	Amsterdam
Organization	Vrije Universiteit
OrganizationalUnit	Math. & Comp. Sc.
CommonName	Main server
Host_Name	zephyr
Host_Address	192.31.231.66

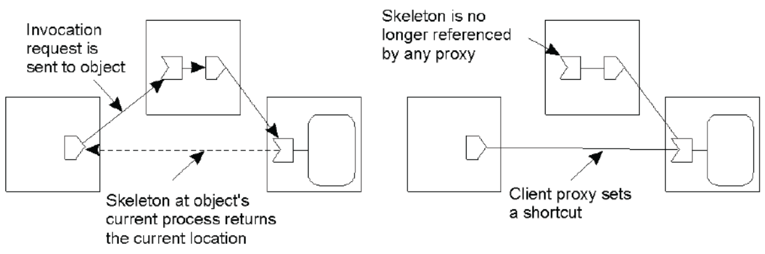


Location Service

- Naming versus Location Entities

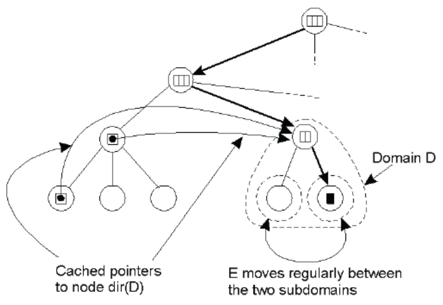


- Forwarding pointers
  - Pointers are forwarded by using the principle of (proxy, skeleton) pairs
  - Shortcuts are used:

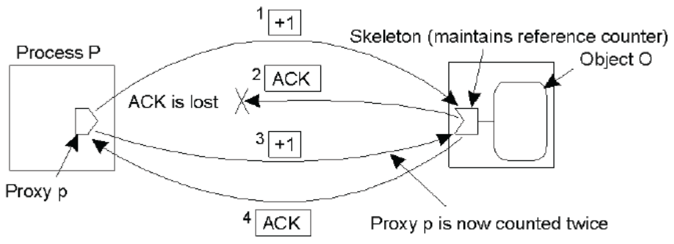

Invocation request is sent to object
Skeleton at object's current process returns the current location
Skeleton is no longer referenced by any proxy
Client proxy sets a shortcut

The same principle is also used in Mobile IP

- Pointer Caches
  - Caching a reference to a directory node of the lowest-level domain in which an entity will reside most of the time

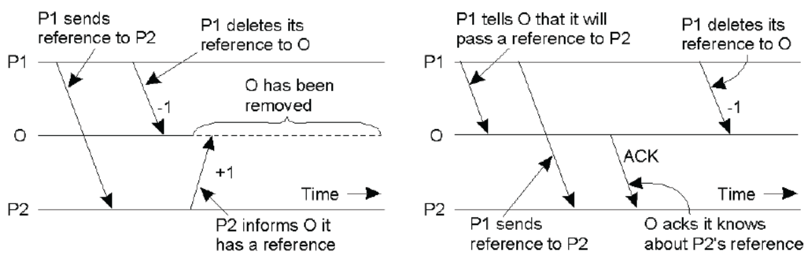

Domain D
Cached pointers to node dir(D)
E moves regularly between the two subdomains

- Scalability
  - The scalability issues related to uniformly placing subnodes of a partitioned root node across the network
- Unreferenced Objects
  - Problem: unreachable entity from the root set
  - Solution: reference counting


1 +1
Skeleton (maintains reference counter)
Object O
Process P
2 ACK
ACK is lost
3 +1
Proxy p
4 ACK
Proxy p is now counted twice

It's difficult to maintain a proper reference count in the presence of unreliable communication

- Problem & Solution of incrementing the reference counter too late:


P1 sends reference to P2
P1 deletes its reference to O
P1
O has been removed
-1
O
+1
Time
P2
P2 informs O it has a reference
P1 tells O that it will pass a reference to P2
P1 deletes its reference to O
P1
-1
O
ACK
Time
P2
P1 sends reference to P2
O acks it knows about P2's reference

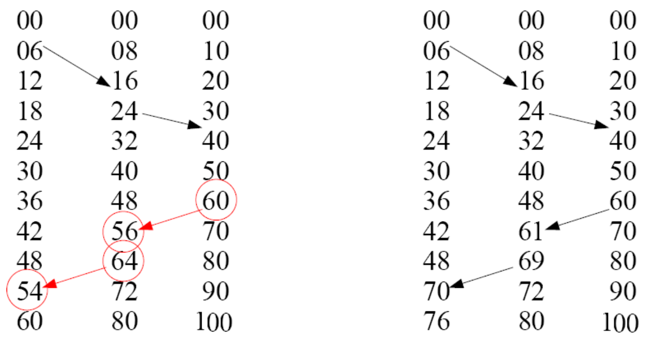
Synchronization

Introduction

- It is necessary to synchronize because
    - Synchronize with respect to time
    - Not access a resource simultaneously
    - Agree on ordering of (distributed) events
    - Appoint a coordinator

Time synchronization

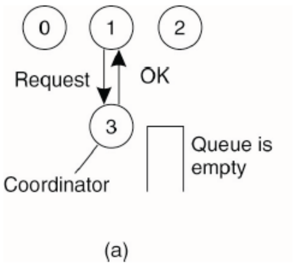
- Synchronization with a Time Server
    - A time server has very accurate time
    - Problem: Messages don’t travel instantly → how can a client synchronize with a time server
    - Solution: Cristian’s algorithm → the transmission delay to the server is estimated
    - In Network Time Protocol (NTP) this algorithm is run multiple times, and outlier values are ignored to rule out packets delayed due to congestion or longer paths
- Logical Clocks
    - In many cases absolute time synchronization is not needed → only the order in which events happen is preserved across all computers
- Lamport timestamps
    - “If a and b are events on the same process, then if a occurs before b, CLOCK(a) < CLOCK(b)”
    - “if a and b correspond to the events of a message being sent from the source process, and received by the destination process, respectively, then CLOCK(a) < CLOCK(b), because a message cannot be received before it is sent.”
    - Wenn ein Prozess eine Nachricht empfängt, die logisch später abgesendet als empfangen wurde, korrigiert der Empfänger seine lokale Uhr, indem er den Zähler um mindestens 1 weitersetzt, als den Zeitstempel in der Nachricht.

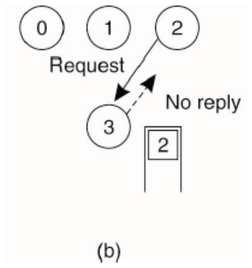


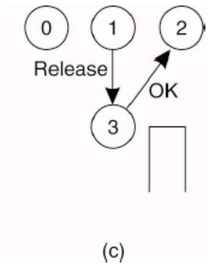
Mutual Exclusion

- Requirements
  - Safety: at most one process may execute in Critical Section at once
  - Liveness: requests to enter and exit the critical section should eventually succeed (no deadlocks or livelocks should occur, and fairness should be enforced)
  - Ordering: requests are handled in order of appearance

Centralized Approach







⊕ Easy to implement

⊕ Few messages necessary (3 per CS: Request, OK, Release)

⊕ Fair (first in first out)

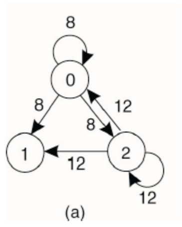
⊕ No starvation

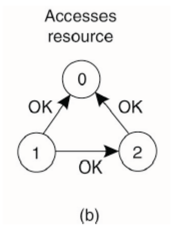
Θ Single point of failure

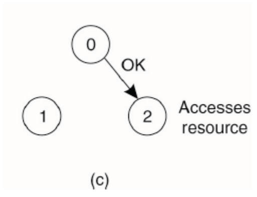
Θ Processes cannot distinguish between dead coordinator or busy resource

Distributed Approach (Ricart & Agrawala's algorithm)

- When a node wants to enter a CS it sends a message with its time and the CS name to all other nodes
- When a node receives such a request
  - If it is not interested in this CS, it replies OK immediately
  - If it is interested in this CS
    - If its message's timestamp was older, then replies OK
    - Else, it puts the sender in a queue and doesn't reply anything (yet)
  - If it is already in the CS, it puts the sender in queue and doesn't reply anything (yet)
- A node enters the CS when it received OK but all other nodes
- A node that exits the CS, sends immediately OK to all nodes that it may have placed in the queue







a) Nodes 0 and 2 express interest in the CS almost immediately
b) Node 0's message has an earlier timestamp, so it wins. Node 1 (not interested) and node 2 (interested, but higher timestamp) send Ok to node 1, so node 1 enters the CS
c) When node 1 exits the CS, it sends OK to node 2, who enters the CS then

- ⊖ More messages: 2 * (n – 1)
- ⊖ No single point of failure but n points of failure. A failure on any one of the n processes brings the system down

Maekawa's algorithm improves: don't wait for approval from all nodes, but from the majority

Token-Ring Approach

- Nodes are organized in a ring and a token goes around
- If a node wants to enter a CS, it can do so when it gets the token
- When it exits the CS, it passes the token to the next node
- ⊕ Very simple
- ⊕ No starvation
- ⊖ Message per entry/exit: 1 to infinite
- ⊖ Problem if the token is lost: long delay might mean that the token is lost or that someone is using it

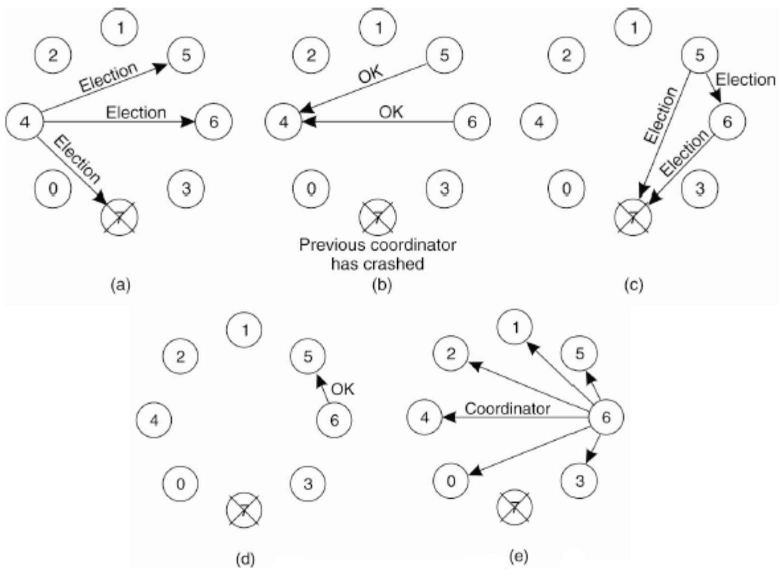
Comparison of these 3 approaches

Algorithm	Messages per entry/exit	Waiting time to enter CS	Problems
Centralized	3	2	Crash of coordinator
Distributed	2*(n-1)	2*(n-1)	Crash of any node
Token ring	1 to infinite	0 to n-1	Lost token, crash of any node

Leader Election

- Choice of one node among a selection of participants
  - Each process gets a unique number
  - Initialize: set all elected(i) = NONE

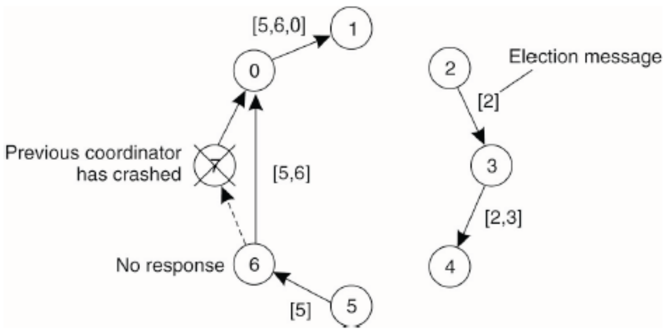
The bully algorithm



- When a node notices that the coordinator is not responding, it starts the election process

- Sends election message to all processes with a higher number; if no response, then it is elected
- If one gets an election message and has higher ID, he replies ok and starts election
- Process that knows it has the highest ID elects itself by sending a coordinator message to all others

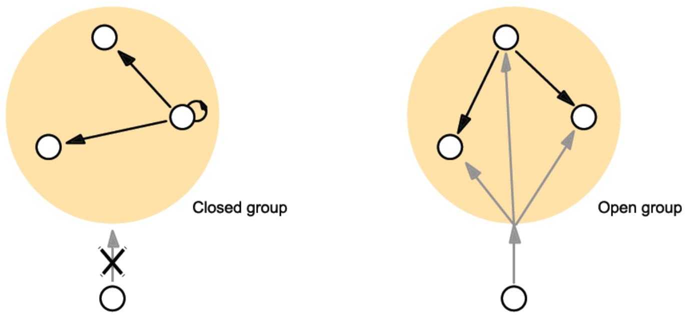
The ring algorithm



- When a node notices that the coordinator is not responding, it starts the election process
- Sends election message to its successor, with a list containing only its own ID
- When one gets an election message that originated at a different node, it appends its ID to the list and forwards the message to its successor
- When one gets back its own election message, it picks the highest ID as the leader and announces it to everyone

The Multicast Problem

- Groups are called closed if only members can send messages



Basic Multicast

- B-multicast(m,g): for each p in g, do send(p,m)
- Problems:
  - Implosion of acknowledgements
  - Not reliable

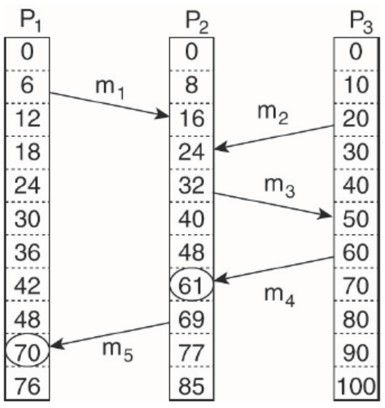
Reliable multicast

- Problems
  - Inefficient: O(g^2) messages
  - Implosion of acknowledgements

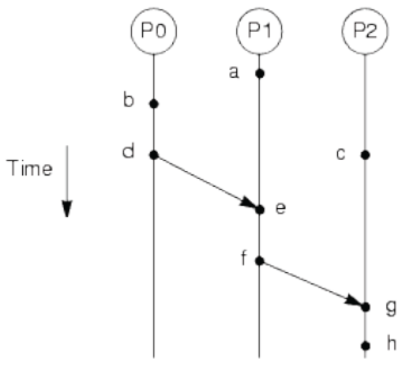
Coordination

Vector Clocks

- Problem: Lamport's timestamp can be used for total ordering of events. However the notion of causality (dependencies between events) is lost

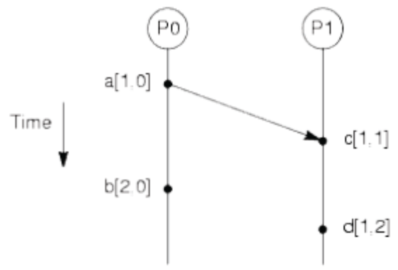


  - The reception of m3(50) could depend on the reception of m2(24) and m1(16). That's correct.
  - The sending of m2(20) seems to be dependent on the reception of m1(16). That's not correct.



  - d consistently has a later timestamp than b, so we would wrongly assume that b → d
- Solution:
  - Each node maintains a vector of N logical clocks
  - One logical clock is its own
  - The rest N-1 logical clocks are estimations for the other nodes
  - Management:
    - They are all initialized by zero
    - When an event happens in a node, it increases its own logical clock in the vector by one
    - When a node sends a message, it includes its whole vector
    - When a node receives a message, it updates each element in its vector by taking the maximum of the value in its own vector clock and the value in the vector in the received message (for every element)
    - An event a is considered to happen before event b, only if all elements of the vector clock of a are less than or equal that the respective elements of the vector clock of b

- Example:



    - a → c because [1,0] < [1,1] and a → d because [1,0] < [1,2]
    - same for a → b and c → d
    - but b and d are independent, because there is no clear order between [2,0] and [1,2]

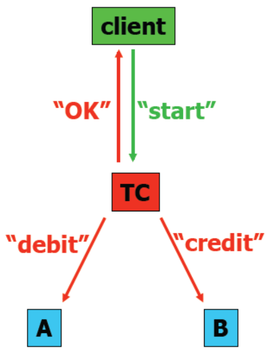
- Implementing at which layer?
  - Middleware layer
    - ⊕ Generic approach
    - Θ Potential (but not definite) causality is captured
    - Θ Some causality may not be captured (external communication can mess up the assumptions of the middleware)
  - Application-specific
    - ⊕ More lightweight
    - ⊕ More accurate
    - Θ Puts the burden of causality checking on the application developer

Atomicity

Guarantee that an operation is completed at all participants (or at none of them)

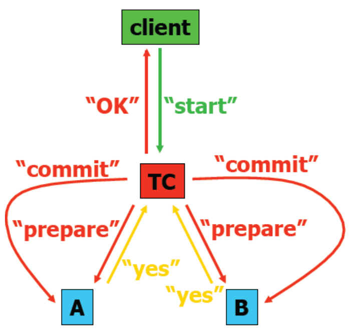
- There are two kinds of atomicity
  - Serializability
    - Series of operations request by users
    - Outside observer sees them each complete atomically in some complete order
    - Requires support for locking
    - For that problem, synchronization (logical/vector clocks) is used!
  - Recoverability
    - Each operation executes completely or not at all
    - No partial results

One-Phase Commit (1PC)



1) Client sends „start“ to TC (Transaction Coordinator)
2) TC sends "debit" to A
3) TC sends "credit" to B
4) TC reports "OK" to client

Two-phase Commit (2PC)



1) TC sends „prepare“ message to A and B
2) A and B respond, saying whether they're willing to commit
3) If both say "yes", TC sends "commit" message
4) If either says "no", TC sends "abort" message
5) A and B "decide to commit" if they receive a commit message

2PC fulfills the correctness property (if one commits, no one aborts and if one aborts no one commits) but not the liveness property (if no failures, and A and B can commit, then commit and if failures come to some conclusion as soon as possible).

- Solution: introduce timeouts and take appropriate actions
    - If TC has not yet sent any "commit" messages it can safely sends "abort" messages

What happens if B (or A) times out?

- If B voted "no" it can unilaterally abort
- If B voted "yes" B and waited too long for an answer, B directly contacts A. It sends "status" request to A, asking if A knows whether the transaction should commit
    - If A received "commit" or "abort" from TC: B decides same way
    - If A hasn't voted anything yet: B and A both abort
    - If A voted "no": B and A both abort
    - If A voted "yes": no decision possible, keep waiting
    - If no reply from A: no decision possible, keep waiting

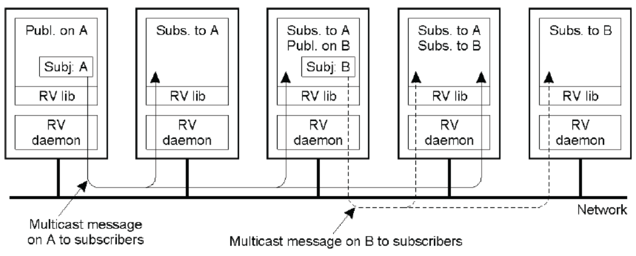
Problem: Crash and Reboot

- Solution: Storing state in non-volatile memory
    - TC writes "commit" to disk before sending
    - A/B write "yes" to disk before sending
    - TC: after reboot, in no "commit" on disk → abort
    - A/B: after reboot, if no "yes" on disk → abort
    - A/B after reboot if "yes" on disk → use ordinary termination protocol (might block!)

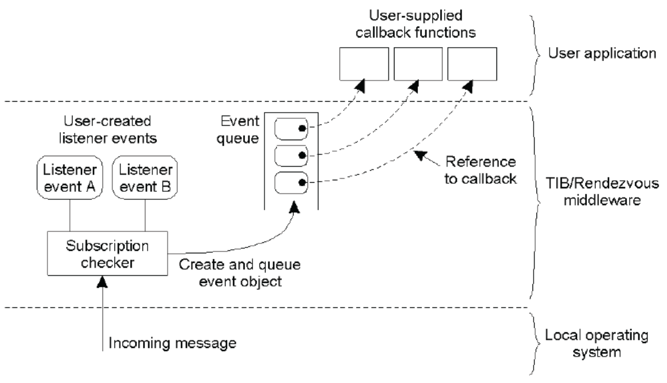
Middleware Systems

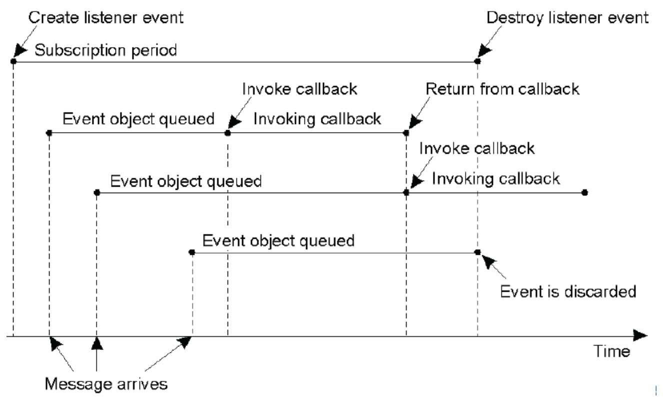
TIB/Rendezvous

- Application dependent communication system
- Messages are self-describing
- Coordination Model

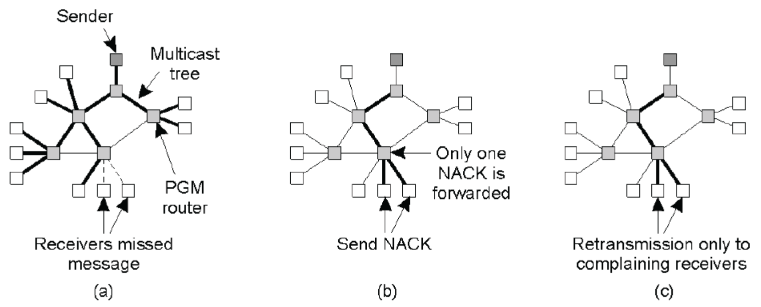


- Event Model:





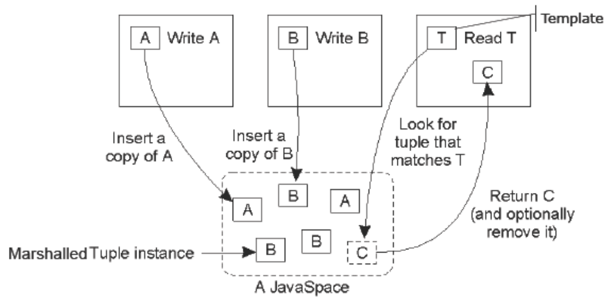
- Reliable Communication:



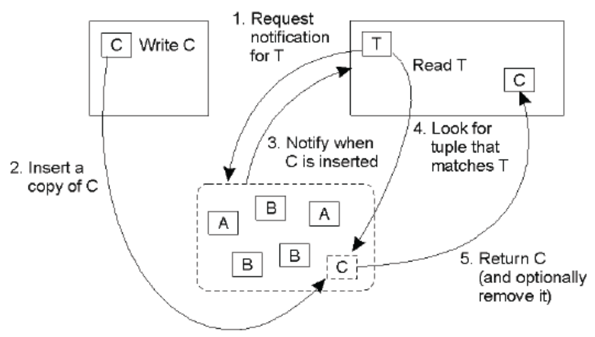
The principle of PGM:
a) A message is sent along a multicast tree
b) A router will pass only a single NACK for each message
c) A message is retransmitted only to receivers that have asked for it.

Jini

Jini ist ein Framework zum Programmieren von verteilten Anwendungen, welche besondere Anforderungen an die Skalierbarkeit und die Komplexität der Zusammenarbeit zwischen den verschiedenen Komponenten stellen und nicht durch existierende Techniken bedient werden können. Jini bietet eine flexible Infrastruktur, über die Dienste (Services) in einem Netzwerk bereitgestellt werden können.

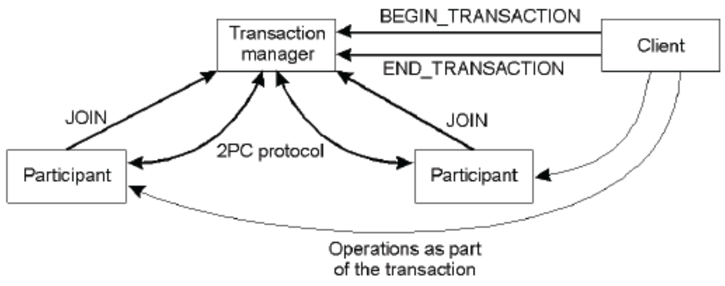


JavaSpaces in Jini

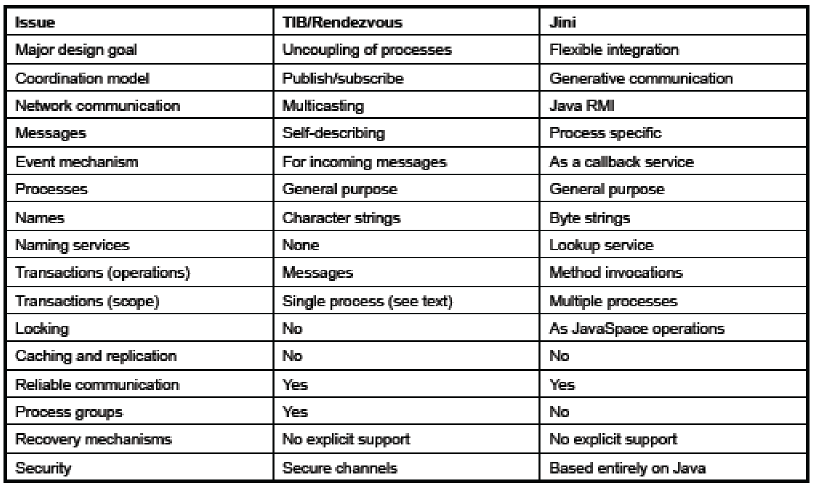

C Write C
1. Request notification for T
T
Read T
C
2. Insert a copy of C
3. Notify when C is inserted
4. Look for tuple that matches T
5. Return C (and optionally remove it)

Communication Events in Jini

Sychronization of Transactions:



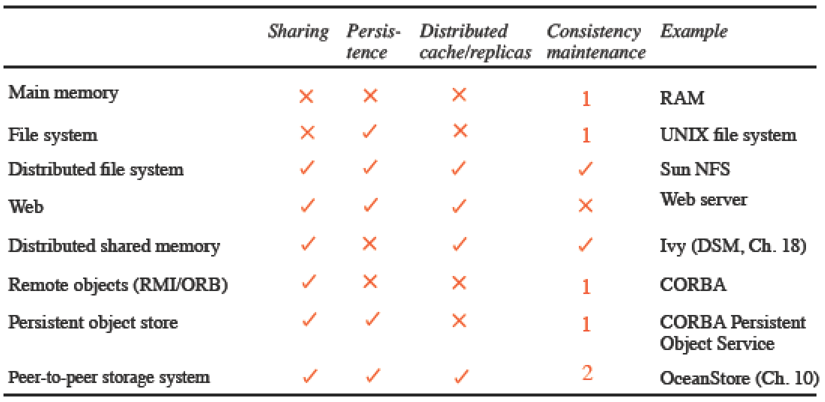
Comparison of TIB/Rendezvous and Jini



Issue	TIB/Rendezvous	Jini
Major design goal	Uncoupling of processes	Flexible integration
Coordination model	Publish/subscribe	Generative communication
Network communication	Multicasting	Java RMI
Messages	Self-describing	Process specific
Event mechanism	For incoming messages	As a callback service
Processes	General purpose	General purpose
Names	Character strings	Byte strings
Naming services	None	Lookup service
Transactions (operations)	Messages	Method invocations
Transactions (scope)	Single process (see text)	Multiple processes
Locking	No	As JavaSpace operations
Caching and replication	No	No
Reliable communication	Yes	Yes
Process groups	Yes	No
Recovery mechanisms	No explicit support	No explicit support
Security	Secure channels	Based entirely on Java

Distributed File Systems

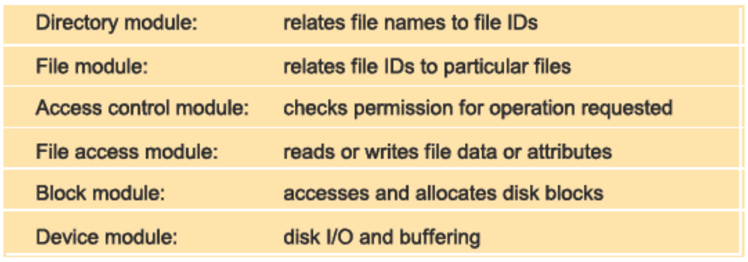
Storage systems and their properties:



	Sharing	*Persis-tence*	*Distributed cache/replicas*	*Consistency maintenance*	*Example*
Main memory	✗	✗	✗	1	RAM
File system	✗	✓	✗	1	UNIX file system
Distributed file system	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	✗	Web server
Distributed shared memory	✓	✗	✓	✓	Ivy (DSM, Ch. 18)
Remote objects (RMI/ORB)	✓	✗	✗	1	CORBA
Persistent object store	✓	✓	✗	1	CORBA Persistent Object Service
Peer-to-peer storage system	✓	✓	✓	2	OceanStore (Ch. 10)

Types of consistency:
1: strict one-copy. 3: slightly weaker guarantees. 2: considerably weaker guarantees.

File system modules:

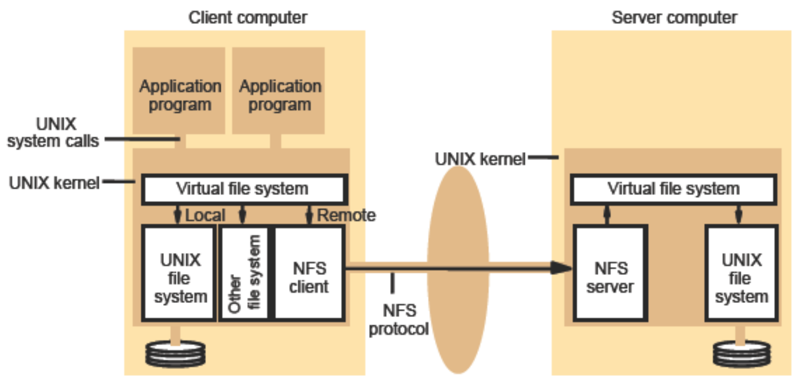


Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	disk I/O and buffering

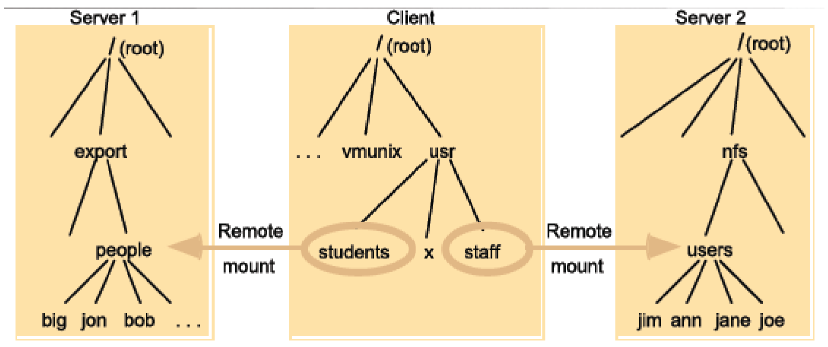
- Distributed File System Requirements
  - Transparency (access, location, mobility, performance and scaling)
  - Concurrent file updates
  - File replication
  - Hardware & software heterogeneity
  - Fault tolerance
  - Consistency
  - Security
  - Efficiency

NFS

Architecture:



Local and remote file systems:

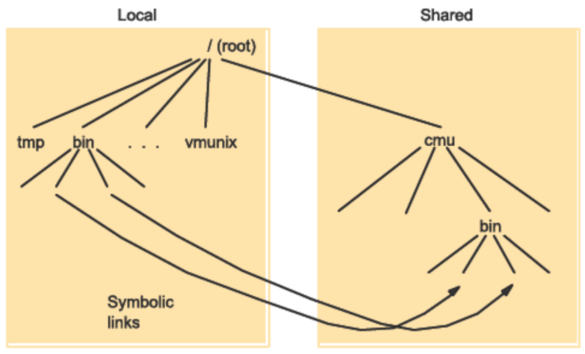


Note: The file system mounted at */usr/students* in the client is actually the sub-tree located at */export/people* in Server 1; the file system mounted at */usr/staff* in the client is actually the sub-tree located at */nfs/users* in Server 2.

Transparency	
Access	NFS client process
Location	File name space identical
Mobility	Remounts possible
Scaling	Experiments are positive
Concurrent file updates	Not supported
File replication	Read-only files
Hard- & Software heterogeneity	Given
Fault tolerance	Stateless and idempotent
Consistency	Close to one-copy semantics
Security	Standard ☹
Efficiency	Seems ok

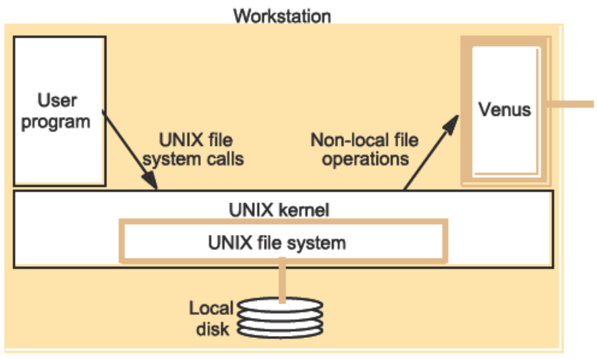
Andrew File System

File name space:



Note: Files are COPIED on access

System call interception in AFS:



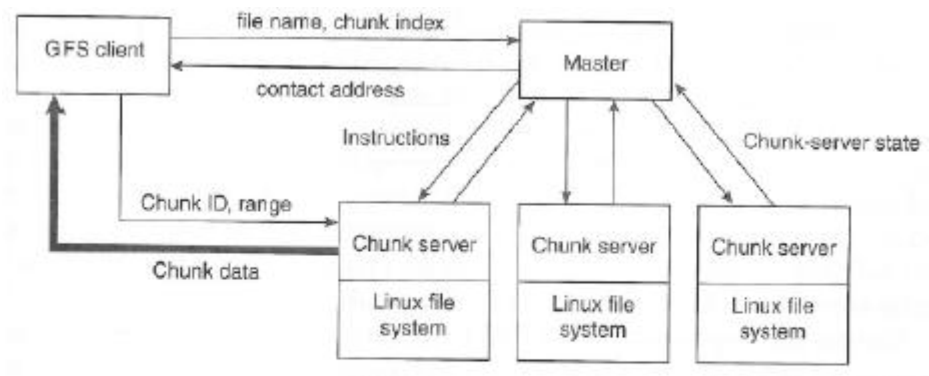
Implementation of file system calls in AFS:

User process	UNIX kernel	Venus	Net	Vice
open(FileName, mode)	If FileName refers to a file in shared file space, pass the request to Venus. Open the local file and return the file descriptor to the application.	Check list of files in local cache. If not present or there is no valid callback promise, send a request for the file to the Vice Server that is custodian of the volume containing the file. Place the copy of the file in the local file system, enter its local name in the local cache list and return the local name to UNIX.	→ ←	Transfer a copy of the file and callback promise to the workstation. Log the callback promise.
read(FileDescriptor, Buffer, length)	Perform a normal UNIX read operation on the local copy.			
write(FileDescriptor, Buffer, length)	Perform a normal UNIX write operation on the local copy.			
close(FileDescriptor)	Close the local copy and notify Venus that the file has been closed.	If the local copy has been changed, send a copy to the Vice server that is the custodian of the file	→	Replace the file contents and send a callback to all other clients holding callback promises on the file.

- Important aspects
  - Explicitly excludes databases
  - UNIX kernel modifications (file system on user level)
  - Location database replicated throughout servers
  - Read-only replicas
  - Bulk transfers (64 KB)
  - Partial file caching
  - Wide area support

Google File System

- Files are distributed over multiple servers in chunks of 64 MB like in a RAID (File Striping)



- Master servers are used as directories only

Peer-to-Peer Systems

Historical Overview

- 1970s & 1980s:
  - Limited reach of the internet
  - Central committee to organize and maintain it
- 1990s:
  - Tremendous expansion and diffusion
  - Killeraps: www and e-commerce
  - Client/server model
- Late 1990s until today
  - P2P as an alternative to client/server
  - End-computers play a role, contribute, interact
- June 1999
  - Napster: Users establish a virtual network, entirely independent of physical network and administrative authorities or restrictions.
  - Basis: UDP and TCP connections between peers
  - Users not only download content, but also provide content to others
- December 1999
  - RIAA lawsuit against the central lookup server but popularity of Napster skyrocketed
- July 2001
  - Napster lost lawsuit → network breaks down immediately

- March 2000
  - Open source project Gnutella: fully decentralized → no single point to attack
- End of 2000
  - Superpeer concept → hierarchical routing layer → significantly improves scalability and efficiency
- 2002
  - BitTorrent started
- 2003
  - BitTorrent caused majority of traffic
  - Downloads significantly faster, due to mechanisms against free-raiding
- Middle of 2003
  - Skype

Define P2P

P2P is a class of systems where:

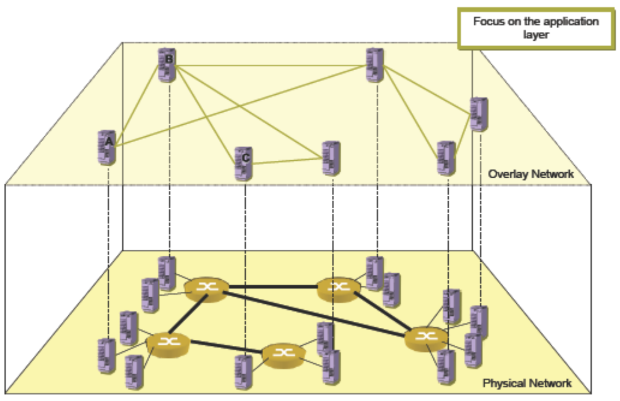
- Resources available at the edges of the internet are utilized (storage, CPU, bandwidth, content, human presence)
- Service is carried out collectively (nodes share both benefits and duties)
- Irregularities and dynamicity are treated as the norm

Main advantages of P2P:

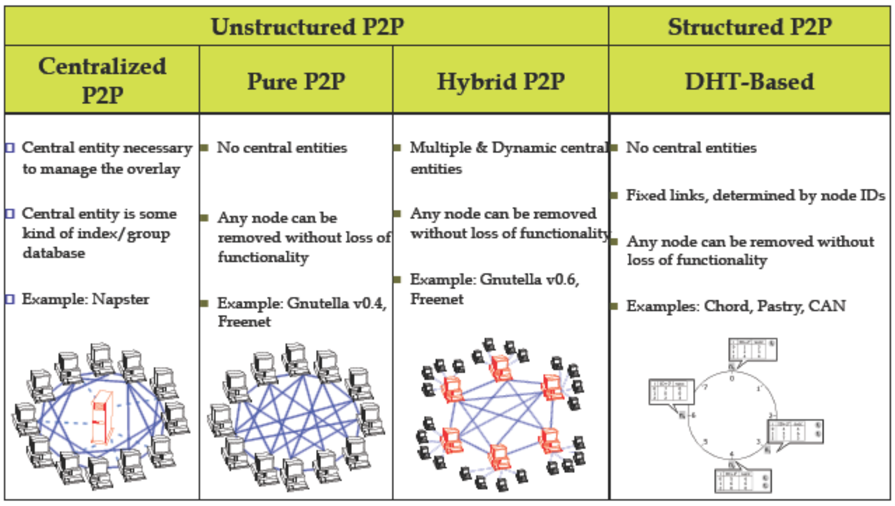
- Inherently scalable (higher demand → higher contribution)
- Increased (massive) aggregate capacity
- Utilize otherwise wasted resource
- Distribute load and administration
- Designed to be fault tolerant
- Inherently handle dynamic conditions

Important issues in P2P

- Overlay networks



Overlay types	
Unstructured P2P	**Structured P2P**
• Any tow nodes can establish a link • Topology evolves at random • Topology reflects desired properties of linked nodes	• Topology strictly determined by node IDs

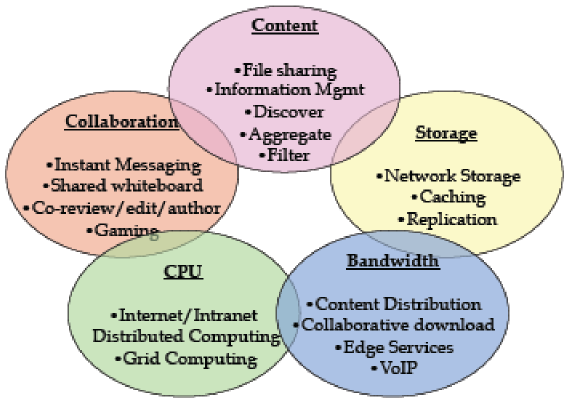


Unstructured P2P			Structured P2P
Centralized P2P	Pure P2P	Hybrid P2P	DHT-Based
Central entity necessary to manage the overlay Central entity is some kind of index/group database Example: Napster	No central entities Any node can be removed without loss of functionality Example: Gnutella v0.4, Freenet	Multiple & Dynamic central entities Any node can be removed without loss of functionality Example: Gnutella v0.6, Freenet	No central entities Fixed links, determined by node IDs Any node can be removed without loss of functionality Examples: Chord, Pastry, CAN

- Overlay maintenance
  - Bootstrapping (how to join the system)
  - Continuous maintenance (how to handle changes and faults)
- Scalability
  - Avoid central server
  - Distribute load on multiple peers
  - Limit load per peer
- Fairness
  - Load balancing
  - User behavior
    - Users are selfish and independent (maximize own benefit)
    - Give incentives for fair play
    - To maximize benefit → abide by the rules
- Dynamicity and adaptability
  - Changing topology because nodes join and leave
  - Changing data (files are added and deleted, content is changed)
  - Changing profiles (user changes interest, new semantic categories introduced)
  - Change in load
- Fault tolerance
  - Robustness of the overlay
  - Resilience to failures
  - Resistance to node & link crashes
  - Availability
- Self-organization
  - No one keeps full state → nodes take local decisions
  - Globally smooth operation should emerge from local decisions
- Performance
  - Efficiency in searching, routing, discovering relationships, etc.
  - Locality → reduce network latency
- Privacy
  - Anonymity
  - Reputation
  - Resistance to censorship

- Security
    - Defend against DDOS attacks
    - Disseminate worm protection patches → speed is crucial
    - Make P2P systems themselves secure
- Legal issues
    - Copyright violation
    - Direct and indirect infringement
- Simplicity

Application Areas



- Sharing content
  - ⊕ large distributed storage
  - ⊕ very high variation of content
  - ⊖ unstable availability
  - ⊖ no guarantees
- Network storage
    - Applications: OceanStore and PAST
- Contributing bandwidth

Client/Server
1. 9h:52m
2. 14h:48m

Source server: 100 Mb/s
Clients: 10 Mb/s
1. Antivirus update
100,000 clients
File: 4 MB
2. Daily database update
1000 clients
File: 600 MB

Cooperative
1. 52s
2. 09m:54s

- Sharing CPU
    - Increasing requirements for High Performance Computing
    - Available computing power of endpoints often unused
    - Use P2P to bundle processor cycles
    - Examples: SETI@home, BOINC, World Community Grid
- Collaboration
    - Presence information (e.g. Instant Messaging Systems)
    - Document collaboration
        - P2P networks which create a connected repository from the local data on the individual peers

- Collaboration
    - Synchronous communication, online meetings, edit shared documents, …
    - P2P Groupware
        - Avoid additional administrative task and central data management
        - All of the data created is stored on each peer and is synchronized automatically

Basics in File Sharing

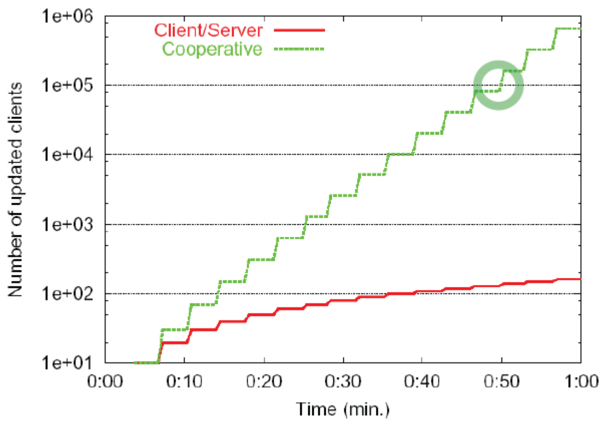
- Napster
    - Relies on a central index but files don't reside on a central server
    - Quick searching (faster and better than Gnutella)
    - Users come and go → user/search database continually updated
    - Automatic file sharing
    - Single server to bring down (this centralization is ultimately its downfall)
- Gnutella
    - Pure P2P
    - Decentralized method of searching (→ harder to "pull the plug")
    - Search by flooding
    - File transfer is direct (→ no anonymity)
    - Problems
        - 70% of people shared no files
        - 50% of search responses from top 1% of hosts
        - Reverting to client/server → suddenly not so hard to shut down
        - Non-standard implementation → some clients are dodgier than others
- Kazaa
    - Hybrid P2P
    - Files and control data are encrypted
    - Everything in HTTP request and response messages
    - Architecture
        - Each peer is either a supernode or is assigned to a supernode
        - Each supernode knows about many other supernodes
        - Supernodes act as mini-Napster hubs tracking the content and IP addresses of their descendants
        - Dedicated user authentication server and supernode list server
    - Overlay maintenance
        - List of potential supernodes included within software download. New peer goes through list until it finds operational supernode

P2P Content Sharing

Motivation behind Decentralized Content Distribution

- A growing number of well-connected users access increasing amounts of content
- Servers and links are overloaded because of the number of clients, the size of content and flash crowds (e.g. 9/11)
- Tremendous costs necessary to make server farms scalable and robust

- Solution: Cooperative Distribution:



  - Principle: utilize bandwidth of edge computers
  - Self-scaling network: more clients → more aggregate bandwidth → more scalability
  - Self-organizing: robust against failures and flash crowds

BitTorrent

- Designed for the transfer of large files to many clients
- Based on swarming: a server sends different parts of a file to different clients, and the clients exchange chunks with one another

Torrent file

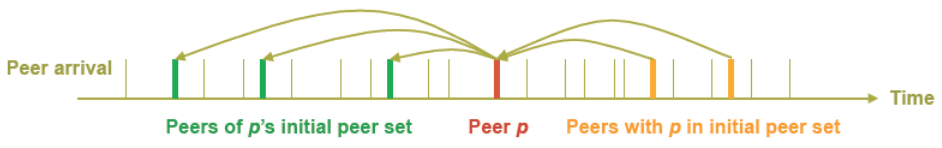
- Tracker address (IP + Port)
- Bytes per chunk
- Number of chunks
- Fear each chunk the SHA1 hash value (helps validate the correctness of downloaded chunks)

Session Initiation

1) Make the torrent file available on a web server
2) The tracker tracks peers
   a. Initially it knows at least one seeder
   b. Matches new peers with existing ones, to allow them collaborate
3) On the client side:
   a. Clients contacts the tracker (over HTTP or HTTPS)
   b. The tracker returns a set of active peers
   c. Clients regularly report state to tracker

Peer Sets

- Tracker picks peer at random on its list
- Once a peer is incorporated in the BitTorrent session, it can also be picked to be in the peer set of another peer → a peer knows both older peers and newcomers
- A peer communicates with its initial peer set and the other peers that contacted it but not with other peer sets



Peer arrival
Time
Peers of p's initial peer set
Peer p
Peers with p in initial peer set

File Transfer Algorithm

- Initially file broken into chunks (typically 256 kB)
- Reports sent regularly (at start-up, shutdown and every 30 minutes) to tracker
- Peers connect with each other over TCP full duplex (data is transit in both directions)
    - Upon connection, peers exchange their list of chunks
    - Each time a peer has downloaded a chunk and checked its integrity, it advertises it to its peer set

Connection States

- “Interesting”: you have a chunk that I want → allows a peer to know its possible client for upload
- “Chocked”: I don’t want to send you data at the time

Chunk Selection Policy

Which missing chunk should we request from other peers?

- Simple strategy: random selection
- Biased strategy: peers apply the rarest-first policy
    - Rare chunks can more easily be traded with others
    - Maximize the minimum number of copies in any given chunk in each peer set

BitTorrent uses rarest-first policy except for newcomers that use random to quickly obtain a first block

Peer Selection Policy

- Seeders’ policy: the ones that offer the best upload rates
- Leechers’ policy: the ones that also serve us: tit for tat

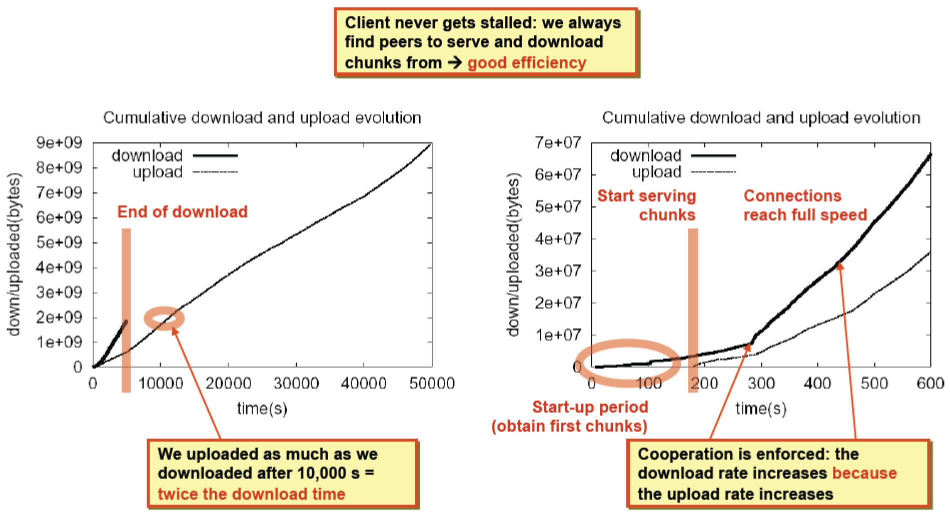
Find better hosts:

- Reconsider choking/unchoking every 10 seconds
- Optimistically unchoke a random peer every 30 seconds to give a chance to another host to provide better service
- Newcomers have less data to offer → give them “priority” in the optimistic unchoke

BitTorrent: Measurements & Evaluation

- Clients’ behavior
    - When they are leechers they have no chois due to tit-for-tat
    - Once download is completed
        - Clients stay on average 3 hours after download
        - The transfer is long, may complete overnight
        - The content is legal (RIAA will not sue)
        - The users are very kind
- Seeders vs. Leechers
    - Presence of seeders is a key feature of BitTorrent
    - Over the 5 months (of the study) they contributed twice as much volume as leechers
- Speed
    - Aggregate throughput of system (sum over all leechers at each instant) was higher than 800 Mb/s

    - This is more than 80 mirrors each sustaining a 10 Mb/s service
    - Nevertheless there is a high variance in download throughputs
- Download and Upload



- Tit-for-tat policy
    - Most of the file provided by peers that connected to us (not from original peer set)
    - Policy must enforce cooperation among peers but also must allow transfer even if bandwidth not perfectly balanced
        - E.g. I don't give you anything because I can send you at 100 kb/s whereas you can only send at 80 kb/s

Distributed Hash Tables

What are DHTs

- Strategies to locate content in P2P
    - Simple strategy: flooding
        - Search cost at least O(N)
        - Need many replicas to keep overhead small
    - Centralized index
        - Single point of failure and high load on this index
    - Indexed search
        - Store particular content on particular nodes
        - When a node wants this content, go to the node that is supposed to hold it
        - Challenges
            - Avoid bottlenecks → distribute the responsibilities evenly
            - Self-organizing w.r.t. nodes joining or leaving
            - Fault-tolerance and robustness
- Hashtables
    - Network has N nodes
    - Each data item has a key
    - Key is hashed to find responsible peer for it
    - Each node is expected to hold 1/N of the items, so that storage is balanced
    - It is also necessary to balance routing load

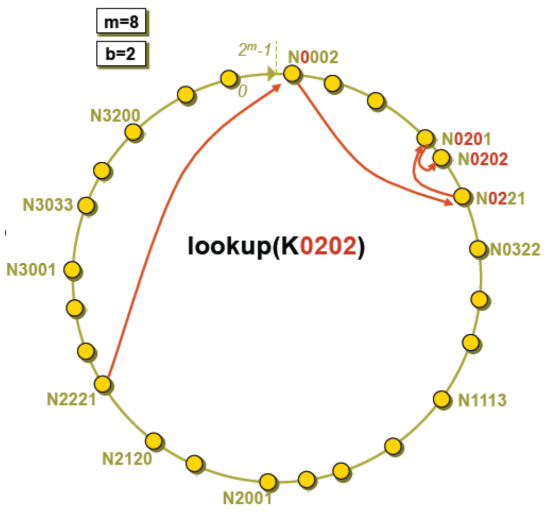
- DHT Design
    - Should be able to route to any node in a few hops (small diameter)
    - DHT routing mechanisms should be decentralized
    - The number of neighbors for each node should remain “reasonable” (small degree)
    - To achieve good performance, DHTs must provide low stretch
    - Should gracefully handle nodes joining and leaving
        - Reorganize neighbor sets
        - Bootstrap mechanisms to connect new nodes into the DHT
        - Repartition the affected keys over existing nodes

Pastry

- Circular m-bit ID space for both keys and nodes
- Addresses have m/b digits
- A key is mapped to the node whose ID is numerically-closest to the key ID

Pastry Routing

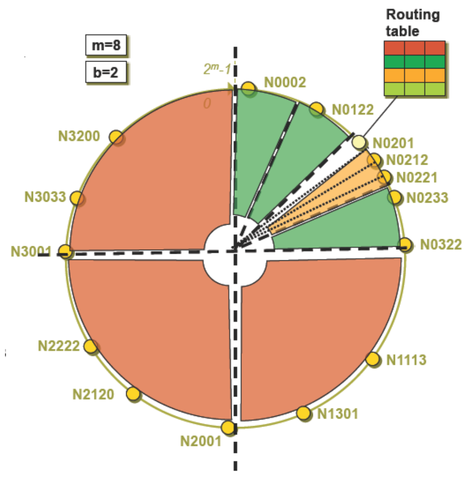
- Can be done in O(log N) hops



Whenever a peer receives a packet to route or wants to send a packet it first examines its leaf set and routes directly to the correct node if one is found. If this fails, the peer next consults its routing table with the goal of finding the address of a node which shares a longer prefix with the destination address than the peer itself. If the peer does not have any contacts with a longer prefix or the contact has died it will pick a peer from its contact list with the same length prefix whose node ID is numerically closer to the destination and send the packet to that peer. Since the number of correct digits in the address always either increases or stays the same — and if it stays the same the distance between the packet and its destination grows smaller — the routing protocol converges.

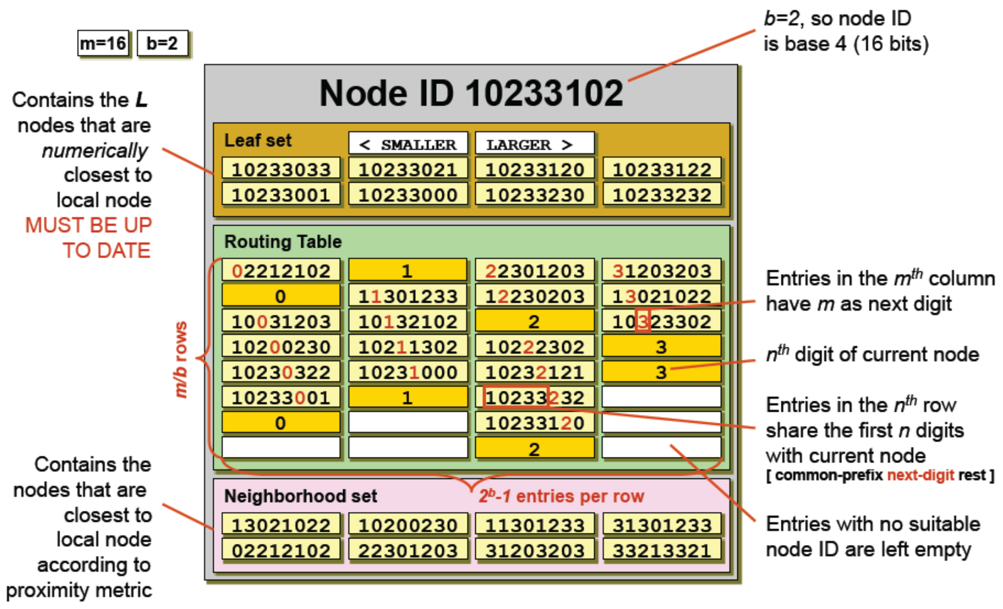
Pastry State and Lookup

- For each prefix, a node knows some other node (if any) with same prefix and different next digit
- When multiple nodes, choose the topologically-closest to maintain good locality properties



N-:	N1???, N2???, N3???
N0:	N00??, N01??, N03??
N02:	N021?, N022?, N023?
N020:	N0200, N0202, N0203

Pastry Routing Table


m=16
b=2
b=2, so node ID is base 4 (16 bits)
Contains the L nodes that are numerically closest to local node
MUST BE UP TO DATE
m/b rows
Contains the nodes that are closest to local node according to proximity metric
Entries in the mth column have m as next digit
nth digit of current node
Entries in the nth row share the first n digits with current node
[common-prefix next-digit rest]
2b-1 entries per row
Entries with no suitable node ID are left empty

Node ID 10233102

Leaf set

	< SMALLER	LARGER >	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

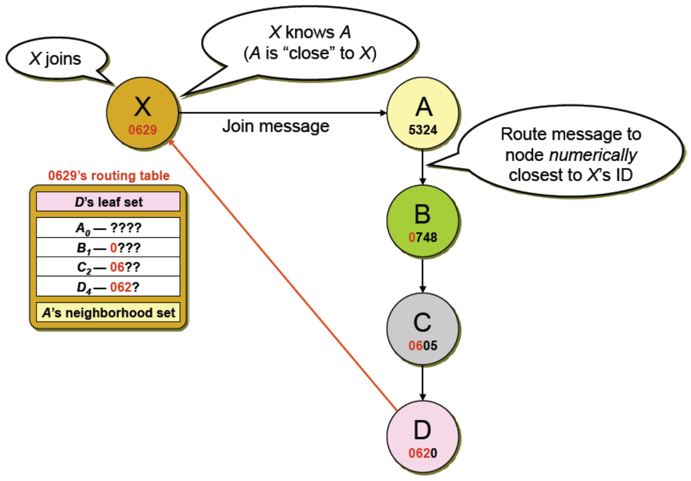
Routing Table

02212102	1	22301203	31203203
0	11301233	12230203	13021022
10031203	10132102	2	10323302
10200230	10211302	10222302	3
10230322	10231000	10232121	3
10233001	1	10233232	
0		10233120	
		2	

Neighborhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

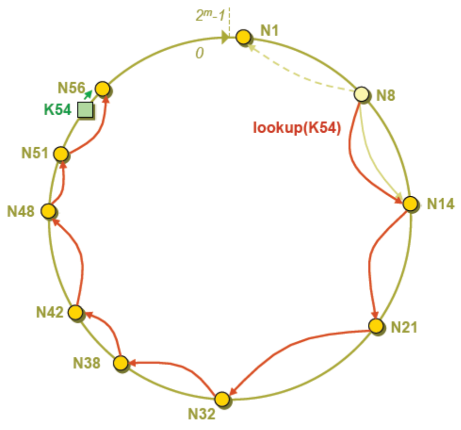
Node Joining



Chord

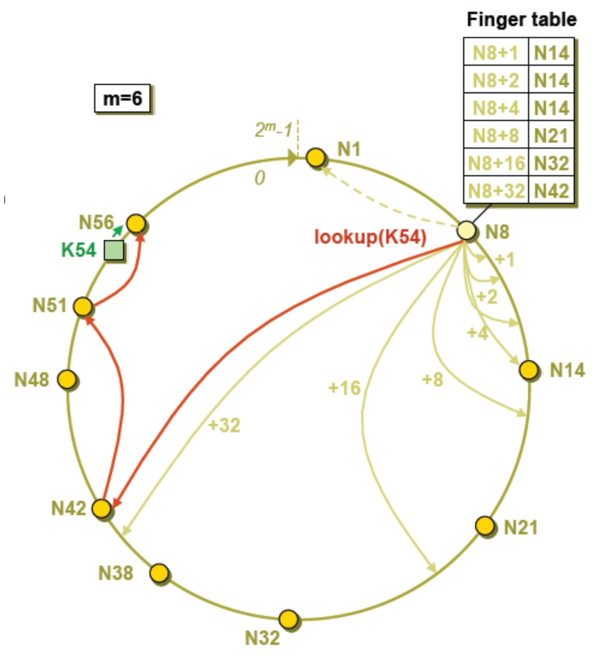
- Circular m-bit ID space for both keys and node IDs
- Each key is mapped to its successor node
- Each node responsible for O(K/N) keys

Lookup in Basic Chord



- Each node knows only two other nodes on the ring
    - Successor
    - Predecessor
- Lookup is achieved by forwarding requests around the ring through successor pointers
- Requires O(N) hops

Lookup in Complete Chord



- Each node knows these two nodes
  - Successor
  - Predecessor
- Each node has m fingers
  - n.finger(i) points to node on or after 2^i steps ahead
  - n.finger(0) == n.successor
  - O(log N) states per node
- Lookup is achieved by following longest preceding finger, then the successor
- O(log N) hops

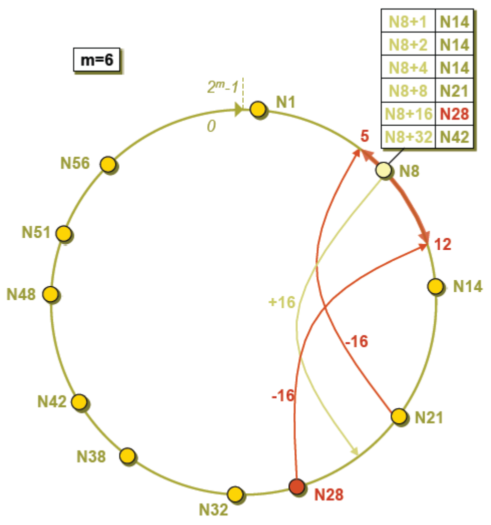
Chord Ring Management

Fingers are for efficiency, not necessarily correctness:

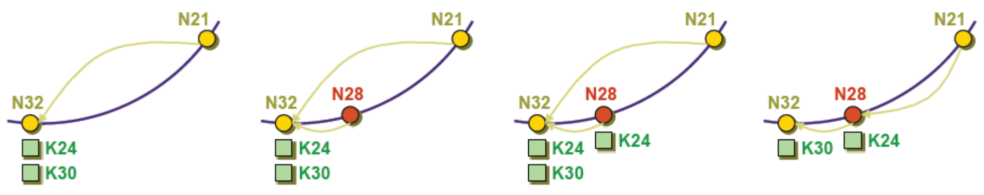
- One can always default to successor-based lookup
- Finger table can be updated lazily

Joining the Ring

1) Initialize predecessor and all fingers of new node j
   a. Locate any node n in the ring
   b. Ask n to lookup the peers at j + 2^0, j+2^1, j+2^2, …
   c. Use results to populate finger table of j
2) Update predecessor and fingers of existing nodes
   a. New node j calls update function on existing nodes that must point to j (nodes in the range [j-2^i, pred(j)-2^i+1])
   b. O(log N) nodes need to be updated



3) Transfer some keys to the new node
    a. Connect to successor
    b. Copy keys from successor to new node
    c. Update successor pointer and remove keys



Leaving the Ring (or failing)

- Node departure are treated as node failures
- Failure of nodes might cause incorrect lookup
- Solution: successor list
    - Each node n knows r immediate successor
    - After failure, n contacts first alive successor and updates successor list
    - Correct successors guarantee correct lookups
- If r = 2 log N, the ring is with a high probability (1-1/N) not broken when half of the nodes fail

Stabilization

- Stabilization algorithm periodically verifies and refresh node pointers (including fingers)

Conclusions

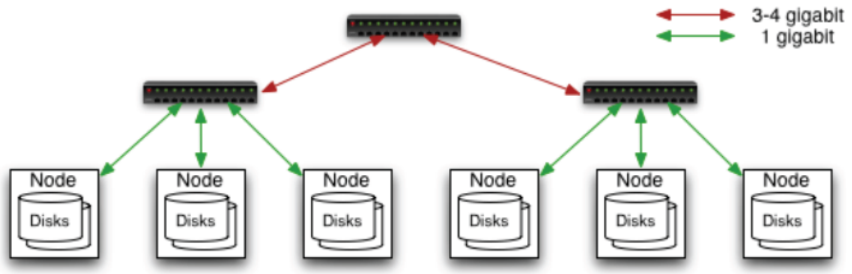
- Search types only allow equality and not range
- Scalability
    - Diameter (search and update): O(log N)
    - Degree: O(log N)
    - Construction: O($\log^2$ N) if a new node joins
- Robustness: replication might be used by storing replicas at successor nodes

Map Reduce

MapReduce ist ein von Google Inc. eingeführtes Framework für nebenläufige Berechnungen über große (mehrere Petabyte) Datenmengen auf Computerclustern. Dieses Framework wurde durch die in der funktionalen Programmierung häufig verwendeten Funktionen map und reduce inspiriert, auch wenn die Semantik des Frameworks von diesen abweicht. MapReduce-Implementierungen wurden in C++, Erlang, Java, Python und vielen anderen Programmiersprachen realisiert.

Hadoop

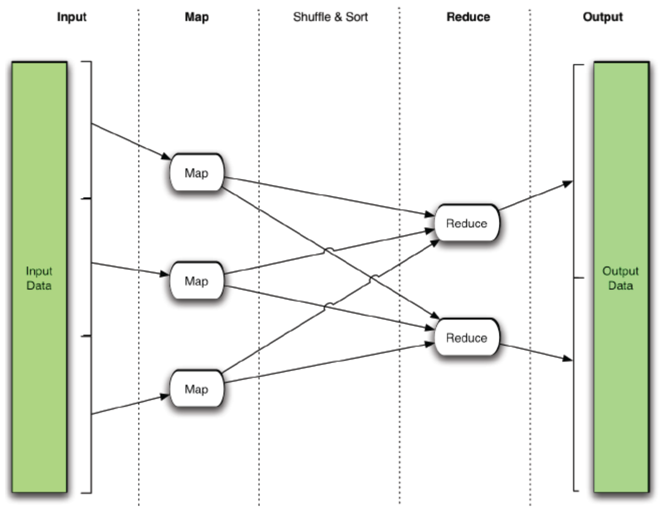
- Open source project written in Java
- Hadoop Core includes
    - Distributed file system → distributes data
    - Map/Reduce → distributes application
- Hardware cluster
    - Typically in 2 level architecture
    - Nodes are commodity PCs



- Distributed file system
    - Single namespace for entire cluster (managed by a single namenode)
    - Files are single-writer and append-only
    - Optimized for streaming reads of large files
    - Files are replicated to several datanodes for reliability
    - Client talks to both namenode and datanodes (but data is not sent through the namenode)
    - Throughput of file system scales nearly linearly with the number of nodes
- File Block placement
    - Blocks are placed on the same node, on a different rack and on the other rack
    - Clients read closest replica
- Data correctness
    - Data is checked with CRC32
    - Validation periodically and on file access

Map/Reduce

- Dataflow



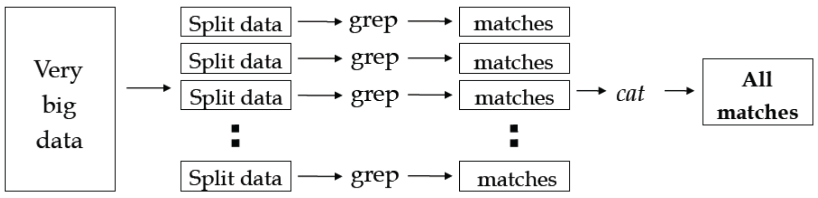
- Features
    - Each task can process data sets larger than RAM
    - Automatic re-execution on failure
    - Locality optimizations (Map tasks are scheduled close to the inputs when possible)
- How is Yahoo using it?
    - Build a huge data warehouse with many Yahoo! Data sets
    - Couple it with a huge computer cluster and programming models to make using the data easy

- Provide this as a service to the researchers → experiments can be run much more quickly with this environment
- Examples
    - Search needs a graph of the “known” web
    - NY Times scanned offline conversion of public domain articles from 1851-1922 (Hadoop was uses to convert scanned images to PDF)
    - Terabyte Sort Benchmark (by Microsoft) was won by Hadoop

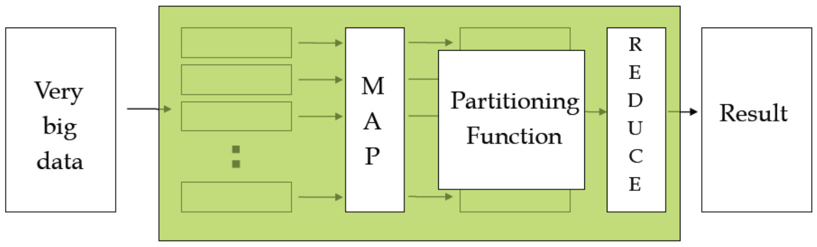
- Further issues
    - Better scheduling
    - Splitting core into subprojects
    - Security
    - High availability

Google MapReduce

Example: Distributed Grep



Functionality



- Map
    - Accepts input key/value pair
    - Emits intermediate key/value pair
- Reduce
    - Accepts intermediate key/value pair
    - Emits ouput key/value pair

Suitable for your task if

- Have a cluster
- Working with large dataset
- Working with independent data (or assumed)
- Can be cast into map and reduce