Advanced Data Management Technologies

Unit 19 — Distributed Systems

J. Gamper

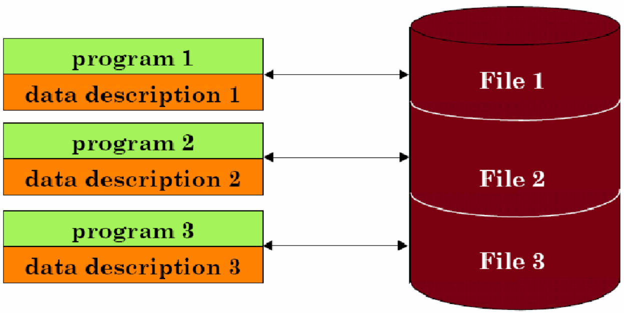
Free University of Bozen-Bolzano
Faculty of Computer Science
IDSE

Outline

Outline

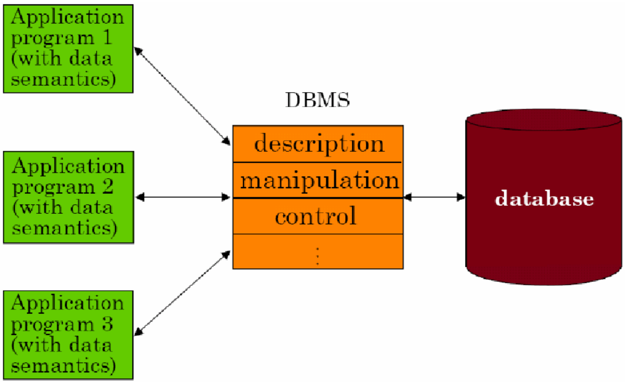
Data Independence/1

- In the old days, programs stored data in regular files
- Each program has to maintain its own data
  - huge overhead
  - error-prone



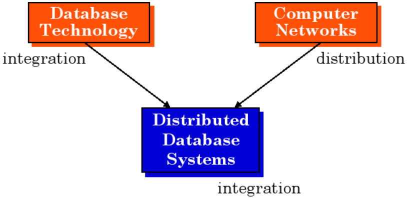
Data Independence/2

- The development of DBMS helped to fully achieve data independence (transparency).
- Provide centralized and controlled data maintenance and access.
- Application is immune to physical and logical file organization.



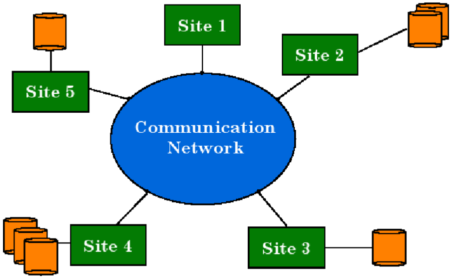
Data Independence/3

- Distributed (database) systems are the union of what appear to be two diametrically opposed approaches to data processing: database systems and computer networks
  - Computer networks promote a mode of work that goes against centralization
- Key issues to understand this combination
  - The most important objective of DBs is integration not centralization.
  - Integration is possible without centralization
- Goal of distributed (database) systems
  - Achieve data integration and data distribution transparency



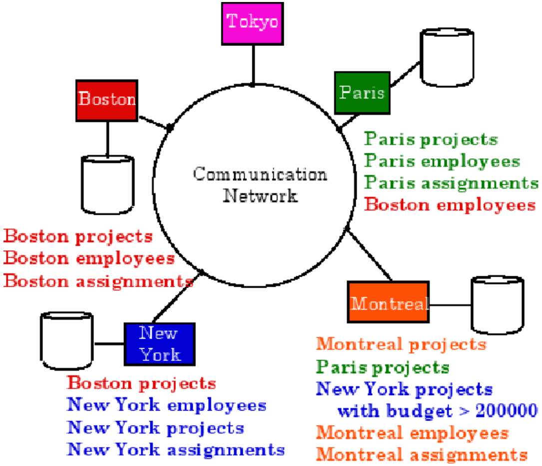
Distributed System

- A distributed (computing) system is a collection of autonomous processing elements (also termed nodes or sites) that are interconnected by a computer network.
- We assume a shared nothing architecture
  - The nodes communicate via message passing (i.e., pieces of data conveying information)
  - They do not share storage or computing ressources



Distributed Database System Example

- Database consists of 3 relations `employees`, `projects`, and `assignment` which are partitioned and stored at different sites (fragmentation).



Promises of Distributed Systems

- Distributed Database Systems deliver the following advantages:
  - Higher reliability
  - Improved performance and scalability
  - Easier system expansion
  - Transparency of distributed and replicated data

Promises – Higher Realiability

- Replication of components
- No single points of failure
  - e.g., a broken communication link or processing element does not bring down the entire system
- Distributed transaction processing guarantees the consistency of the database and concurrency.

Promises – Improved Performance and Scalability

- Proximity of data to its points of use
  - Reduces remote access delays
  - Requires some support for fragmentation and replication
- Parallelism in execution
  - Inter-query parallelism
  - Intra-query parallelism
- Update and read-only queries influence the design of DDBSs substantially
  - If mostly read-only access is required, as much as possible of the data should be replicated
  - Writing becomes more complicated with replicated data

Promises – Easier System Expansion

- Issue is scalability for huge amounts of data
- Emergence of commodity computers and workstation technologies
  - Network of workstations much cheaper than a single mainframe computer

Promises – Transparency

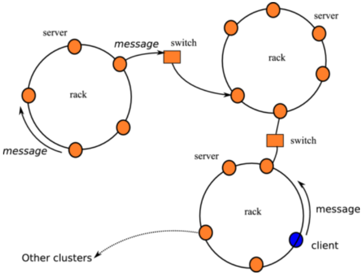
- Refers to the separation of the higher-level semantics of the system from the lower-level implementation issues
- A transparent system hides the implementation details from the users and provides a high-level interface for the development of complex applications.
- Various forms of transparency can be distingushed:
  - Network transparency
    - Location transparency
    - Naming transparency
  - Replication transparency
  - Fragmentation transparency
  - Transaction transparency
    - Concurrency transparency
    - Failure transparency
  - Performance transparency

Outline

Physical Networks

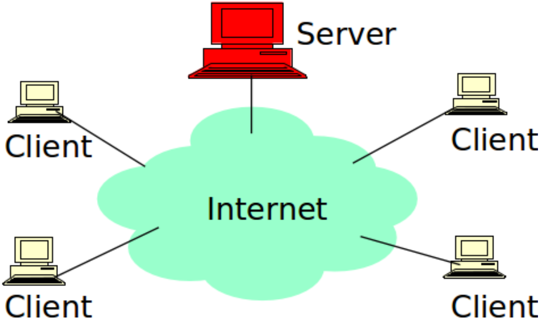
- Local area networks (LAN) are used in data centers to connect hundreds or tousands of servers
- The Internet (a WAN) links millions of LANs
- 3 communication levels can be distinguished:
  1. Servers are grouped on "racks", linked by a high-speed cable. A typical rack contains a few dozens of servers.
  2. A data center consists of (possibly a large number of) racks connected by routers (or switches) that transfer non-local messages.
  3. A (slower) communication level between distinct clusters, e.g., to allow independent data centers to cooperate.

- In 2010, a typical Google data center consists of 100–200 racks, each hosting ≈ 40 servers.
- Today, the number of servers is above one million.



Client/Server Architecture

- A client/server architecture is a particular kind of overlay network on top of a physical network (e.g., the Internet)
- A reliable server is a data source
- Clients request data from server
- Well known and very successful model in some domains
  - WWW (HTTP), FTP, Web services, etc.



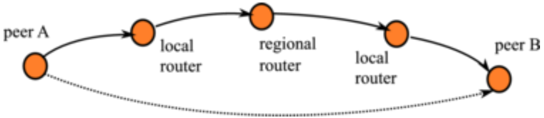
Limitations of Client/Server Architecture

- Scalability is hard to achieve
- Server presents a single point of failure
- Requires administration
- Unused resources at the network edge

- P2P systems try to address these limitations

Peer-to-Peer Networks/1

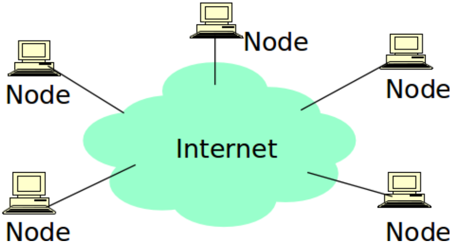
- A P2P network/system is a particular kind of overlay network, a graph structure build over a native physical network.
- Nodes are called peers and communicate with messages sent over the Internet.



- Typically, a message sent by peer A first reaches a local router, that forwards the message to other routers (local, regional, or world-wide) until it is delivered to peer B.
- By abstracting this complexity, a P2P network imagines a direct link between A and B, as if they were directly connected.
- This pseudo-direct connection that may (physically) consist of 10 or more forwarding messages, or hops, is called an overlay link.

Peer-to-Peer Networks/2

- All nodes/peers are both clients and servers
- Nodes provide and consume data, content, storage, memory, CPU
- Nodes are autonomous, i.e., no administrative/centralized authority
  - "The ultimate form of democracy on the Internet"
- Any node can initiate a connection
- Nodes collaborate directly with each other (not through servers)
- Network is dynamic: nodes enter and leave the network "frequently"
- Nodes have widely varying capabilities



Benefits of P2P Systems

- Scalability
  - Consumers of resources also donate resources
  - Aggregated resources grow naturally with utilization
- Reliability
  - Replicas
  - Geographic distribution
  - No single point of failure
- Ease of administration
  - Nodes are self organized
  - Built-in fault tolerance, replication, and load balancing
- Efficient use of resources

Unstructured P2P Networks

- Unstructured P2P networks do not impose a particular structure on the overlay network, but are formed by nodes that randomly form connections to each other, e.g., Gnutella, Gossip, and Kazaa.
- Due to the lack of structure, flooding is the only search technique:
  - Peer disseminates request to all its friends, which flood in turn their own friends, and so on until the target of the request is reached.
  - Flooding is limited by a "Time to live" (TTL) bound: number of times a query is forwarded before being discarded to avoid using too much resources.
- Simple and easy to build as a peer only needs to know some friends to join a network.
- No guarantee that flooding finds the desired data
  - in particular for rare data shared by only a few peers it is very unlikely
- Not very efficient and inherently unstable
  - Peers are autonomous and selfish, yielding frequently a very high rate of peers going in and out of the system.
  - It is difficult to guarantee that a node stays connected to the system, or that the overall topology remains consistent.

Structured P2P Networks

- In structured P2P networks the overlay is organized into a specific topology following a specific protocol.
- This provides more structured ways of looking up the network and to avoid the blind and uncontrolled flooding mechanism.
- The protocol ensures that any node can efficiently search the network for data, even if the data is extremely rare.
- Joining the network becomes more involved as nodes have to satisfy certain criteria.
- BUT, improved performance and stability.
- Distributed Hash Tables (DHTs) are the most popular search mechanism in structured P2P networks (see next unit).

Latency and Bandwidth

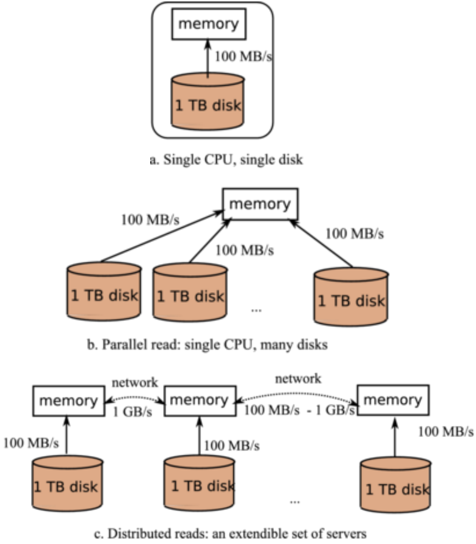
- Different network latency and bandwidth are encountered in P2P systems.
- Both parameters have a huge impact on the performance in P2P systems.

Type	Latency	Bandwidth
Disk	≈ 5 ms	at best 100 MB/s
LAN	1–2 ms	1 GB/s (single rack), ≈ 100 MB/s (switched);
Internet	Highly variable: 10–100 ms	Highly variable: typical a few MBs

- Test these values on your own infrastructure by using
  - `ping` or
  - Web sites, e.g., `http://www.pcpitstop.com/internet/Bandwidth.asp`

Distributed Storage Systems Example

- Read 1 TB of data
- Sequential access: 2,5 hours
- Parallel access: 1 TB spread over 100 disks, all on the same machine
  - Read 10 GB from each disk
  - 1,5 min if all disks work in parallel
  - CPU overloaded if size of data increases
- Distributed access: 100 computers, each with local disk
  - Same disk-memory transfer time
  - But, CPU is not overloaded.



Performance of Distributed Storage Systems

- Disk transfer rate is a bottleneck for batch processing of large scale data sets.
  - Parallelization and distribution of the data on many machines is a means to eliminate this bottleneck.
- Disk seek time is a bottleneck for transactional applications (point queries) that submit a high rate of random accesses.
  - Replication, distribution of writes and distribution of reads are the technical means to make such applications scalable.
- Data locality: whenever possible, program should be "pushed" near the data they need to access to avoid costly data exchange over the network.

Outline

Data Replication and Consistency

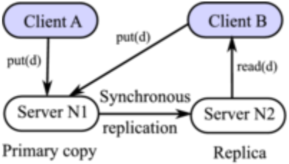
- Data replication is at the core of distributed systems, as most of the properties of distributed systems depend on it.
  - Without replication, the loss of a server hosting a unique copy of some data item results in unrecoverable damages.
  - Ability to distribute read/write operations for improved scalability.
- Problems raised by data replication
  - Performance: writing several copies of an item takes more time, which may affect the throughput of the system.
  - Consistency: consistency management becomes difficult in a distributed setting.

Replication Policies

- Replication policies consider the interactions between performance and consistency issues
- Different technical choices:
  - eager (synchronous) or lazy (asynchronous) replication
  - primary or distributed versioning
- This gives four different replicaton policies

Eager/Synchronous Replication with Primary Copy

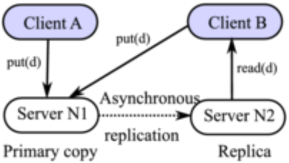
- A put(d) request sent by Client A to Server N1 is replicated at once on Server N2.
- The request is completed only when both N1 and N2 have sent an acknowledgment
  - meanwhile, A is frozen, as well as any other client that would access *d*
- Each data item has a primary copy and several (at least one) secondary copies
- Each update is first sent to the primary copy



- **Properties**
  - \+ A read request sent by client B always accesses a consistent state of *d*, whether it reads from server N1 or N2
  - \+ Requests sent by several clients relating to the same item *d* can be queued, which ensures that updates are applied sequentially and not in parallel
  - – The obvious downside is that these applications have to wait for the completion of other clients' requests, both for writing and reading

Lazy/Asynchronous Replication with Primary Copy

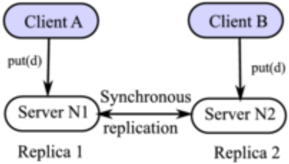
- There is still a primary copy, but the replication is asynchronous
- Some of the replicas may be out of date with respect to client's requests
  - e.g., client B may read from server N2 an old version of item *d* because the synchronization is not yet completed
- Often termed "Master-slave" replication



- **Properties**
  - \+ Client has never to wait
  - − Client might read an old version of data. However, due to the primary copy, the replicas will eventually be consistent because there cannot be independent updates of distinct replicas.
    - Considered acceptable in many modern "NoSQL" data management systems that accept to trade strong consistency for a higher read throughput

Eager/Sychronous Replication without Primary Copy

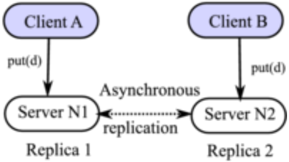
- No primary copy anymore, but eager replication
- Two clients can simultaneously write on distinct replicas
- BUT, the eager replication implies that these replications must be synchronized right away



- **Properties**
  - + Inconsistencies are avoided
  - – It is likely to get some kind of interlocking, where both clients wait for some resource locked by another one

Lazy/Asynchronous Replication without Primary Copy

- Both primary copies and synchronous replication are given up
- Most flexible form of replication
- Often referred to as "Master-Master" replication



- **Properties**
  - + Client operations are never stalled by concurrent operations (optimistic approach, lock-free)
    - Often decisive for Web-scale data intensive applications
  - – Possibly inconsistent states
    - Management of inconsistent replicas required (data reconciliation)
    - Practical approach often used: promote one version as "current" and inform others about a conflict, e.g., CVS, SVN

Different Consistency Levels

- Data replication leads to several consistency levels
  - Strong consistency (ACID properties)
    - Requires a (slow) synchronous replication, and possibly heavy locking mechanisms
    - Traditional choice of database systems
  - Eventual consistency
    - Trades eager replication for performance
    - The system is guaranteed to converge toward a consistent state (possibly relying on a primary copy)
  - Weak consistency
    - Chooses to fully favor efficiency, and never wait for write and read operations
    - Some requests may serve outdated data
    - Inconsistencies typically arise and the system relies on reconciliation based on the application logic

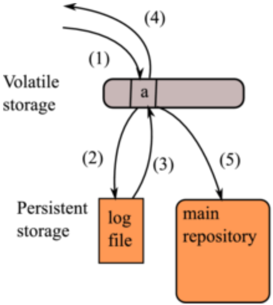
Outline

Failure Management

- Centralized system
  - If a program fails, the simple (and standard) solution is to abort and then restart its transactions
  - Chances that a single machine fails are low
- Distributed system with thousands of commodity computers
  - Failures are quite frequent due to program bugs, human errors, hardware or network problems, etc.
  - Small tasks: simplest solution to restart them.
  - Long lasting distributed tasks: restarting a whole transaction is often not an acceptable option, since
    - errors typically occur too often and
    - in most cases a failure affects only a minor part of the task

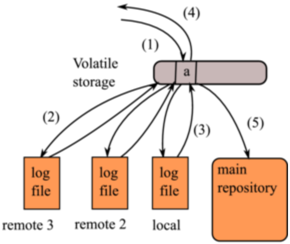
Failure Reovery in Centralized DBMSs

- Standard recovery in centralized DBMSs are based on a persistent log file
  - Client issues a `write(a)` (1)
  - The server does not write immediately *a* in its repository, because a random access is too inefficient
  - Instead, the server writes *a* in an append-only log file (2), which is efficient
  - When the log manager confirms that the data is indeed on persistent storage (3), the server can send back an acknowledgment to the client (4)
  - Eventually, the main memory data will be flushed in the repository (5)
- Recovery is possible from the log file (REDO protocol)



Failure Recovery in Distributed Systems

- Server must log a write operation to the local log file (3) and to one or more remote logs (2)
- Depends on the use of either a synchronous or asynchronous protocol (similar to replication policies).

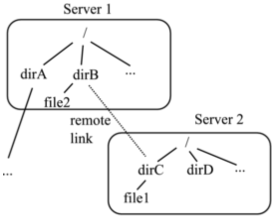


  - Synchronous protocol
    - Client waits for slowest writer, i.e., server acknowledges the client (4) only when all remote nodes have sent a confirmation of successful write operation
    - This may severely hinder the efficiency of updates
    - But, all replicas are consistent
  - Asynchronous protocol
    - Client waits only for fastest writer, i.e., until the fastest copy has been written
    - Puts a risk on data consistency, as a subsequent read operation may access an older version that does not yet reflect the update
- Recovery
  - If the server dies, the closest mirror can be chosen
  - It reads from its own log a state equivalent to that of the dead server, and can begin to answer client requests

Outline

Why a New File System for Very Large Data?

- NFS (in the UNIX world) provides a standard solution to share files among computers
- Assume that server 1 needs to access the files located in the directory `dirC` on server 2
- NFS allows `dirC` to be "mounted" in local FS
- User can navigate to the files stored in `/dirB/dirC` just as if it was fully located on the local computer (transparent name space)

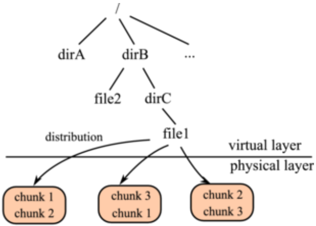


A traditional network file system

- NFS is not designed for very large scale, data-intensive applications and breaks some principles.
  - Does not provide data locality
    - A process on server 1 in charge of manipulating data on server 2 will strongly stress the network bandwidth
  - The approach is hardly scalable, there is no load balancing
    - if file1 stores 90% of the data, server 2 will serve 90% of the client requests

A New DFS for Very Large Data

- In a distributed file system (DFS), a file is no longer the storage unit, but is decomposed in "chunks" of equal size, each allocated by the DFS to the participating nodes
- There exists a global file system namespace shared by all nodes in the cluster
  - Defines a hierarchy of directories and files
  - "Virtual" as it does not affect in any way the physical location of its components.

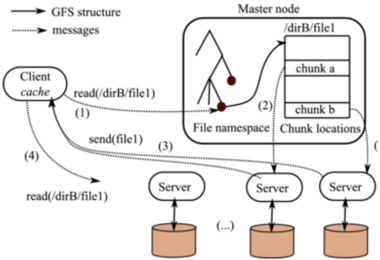


A large scale distributed file system

- Files are mapped in a distributed manner to the cluster nodes
  - e.g., file1 is split in three chunks, each chunk is duplicated, and the two copies are each assigned to a distinct node
- Properties
  - A fair balancing is natively achieved since a file is split in equal-size chunks and evenly distributed
  - Reliability is obtained by replication of chunks
  - Availability can be implemented by a standard monitoring process

The Google File System (GFS)

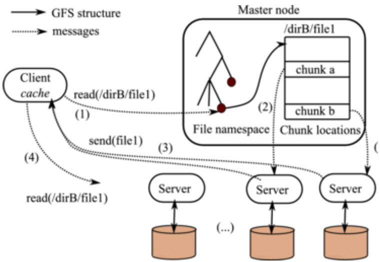
- Consists of a Master node and many server nodes
- Master is the coordinator
  - Receives client connections/requests
  - Maintains the description of the global file system namespace and the allocation of file chunks
  - Monitors system state with "heartbeat" messages in order to detect failures as early as possible



- Servers receive file chunks and must take appropriate local measures to ensure the availability and reliability of their (local) storage

Increasing Scalability of the Google File System

- A single-master architecture brings simplicity, but raises concerns about scalability and reliability
- Client image solves scalability issue: a cache to store meta-information about the location of file chunks



- Example: Client sends a `read(/dirB/file1)` request
  - First request is routed to the Master (1)
  - Master inspects the namespace and finds that file1 is mapped to a list of chunks; their location is found in a local table (2)
  - Each server holding a chunk of file1 transmits this chunk to the client (3)
  - Client keeps in cache the addresses of the nodes that serve `file1`
  - This knowledge can be used for subsequent accesses to `file1` (4)
- Client image avoids a systematic access to the Master for each request
  - By limiting the exchanges with the Master to metadata information, the coordination task is reduced and can be handled by a single computer.

Error Handling in the Google File System

- Failures are handled by standard replication and monitoring techniques
- Chunks are replicated on at least 3 servers
  - Master is aware about the replicas
- If a server does not answer to a heartbeat message, Master initiates a server replacement
  - Ask one of the other servers (with the same replicas) to copy the relevant chunks to a new server.
- Master itself needs special protection because it holds the file namespace
  - A specific recovery mechanism is used for all the updates that affect the namespace structure

Summary

- Distributed system is a collection of autonomouos processing elements (nodes/sites) that are conneced by a network.
- Distributed Systems promise improved realiability, performance, and scalability.
- P2P networks provide a powerful distributed infrastructure
  - Overlay network on top of a physical network.
  - No distinction between client and server (nodes are both)
  - Dynamic and flexible, i.e., nodes can enter and leave the network
  - Structured versus unstructured P2P systems
- Different latency and network bandwith need to be considered P2P systems.
- Data replication is at the core of distributed systems, but raises problems of performance and consistency.
  - Different replication policies lead to different consistency models.
  - In Web scale applications, eventual or weak consistency is often preferred over strong consistency.
- Failure management is based on log-file (similar to centralized systems)
- Distributed file system for very large data
  - File is decomposed into chunks, which are replicated on different nodes.
  - Natively supports a fair balancing, reliability and availability.