

## Lecture 1

## Distributed and Network Programming – Notes – Ahmed Nouralla

- **Distributed System (DS)**

- A collection of autonomous computing elements (hosts) that appears to its users as a single coherent system, they communicate via a network (by sending messages) and collaborate in achieving tasks.
- **Middleware:** works as a unified interface between distributed applications and each host local OS.
- **Reasons to make systems distributed**
  - They are inherently distributed (they utilize network for communication)
  - For better performance, reliability, and for solving big tasks.
- **Characteristics:**
  - Nodes operate concurrently and may share resources.
  - No global clock (coordination is done by message exchange)
  - Nodes may fail independently (may introduce network partitioning)
- **Design goals:**
  - **Support sharing resources**
    - Problems such as race conditions and non-determinism has to be accounted for.
  - **Openness:**
    - System components should be upgradable/replaceable by people who didn't build it, key interfaces should be provided.
    - Communication between nodes has to be implemented through a unified mechanism.
  - **Scalability:** the extent to which the system can expand without failures.
    - Scalability and availability are the most important goals of a DS (originally designed to handle many geographically-distributed users, do more complex computation, store larger amount of data, without overloading any node or congesting the network).

Type	Description
Size	A size-scalable system maintains performance and usability for its users, regardless of how many resources it might have.
Geographical	A geographically-scalable system handles larger distances between nodes without failing.
Administrative	An administrative-scalable system allows adding more users without introducing management problems (admin should still be able to use the system smoothly).

- **Distributed transparency:**

- **Full transparency** is not required and is not the main goal of a DS.

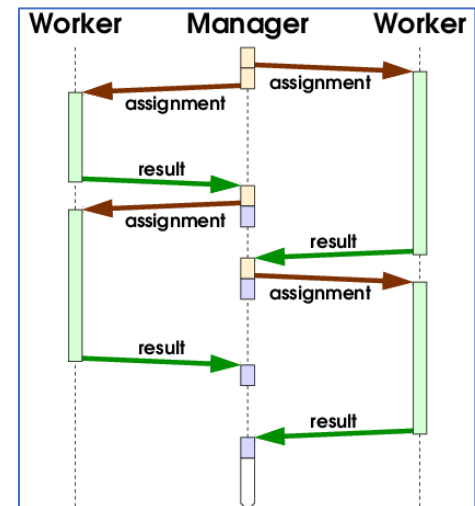
Type	Description
Access	Hiding differences between each node's internal representation of data.
Location	Hide where object is located.
Relocation	Hide the process of <b>moving</b> objects between nodes <b>while in use</b> .
Migration	Hide the process of <b>moving</b> objects between nodes.
Replication	Hide the process of <b>cloning/replicating</b> objects across nodes.
Concurrency	Hide the fact that the object might be shared between users.
Failure	Minimize problems that user experience when system fails and recovers, also try to minimize the number of users affected.

## Lecture 2-3: Recap

- **Recap from previous courses:**
  - **Networks**
    - Introduction, internet and common protocols.
    - TCP/IP and OSI protocol stacks.
    - UDP/TCP comparison
    - Unicasting vs Multicasting
    - Socket programming. (blocking/non-blocking modes).
    - Python `socket` module
  - **OS, Computer Architecture**
    - Processes, Threads, Multithreading, and Concurrency
    - Race conditions, critical section, locks.
    - Producer-Consumer model
    - Python `threading` and `multiprocessing` modules.
    - Global Interpreter Lock (GIL) in Python
      - Allows only one thread to execute code at a time.
      - Other threads can do system calls (e.g., handling I/O)

## Lecture 4: Threading in Network Applications

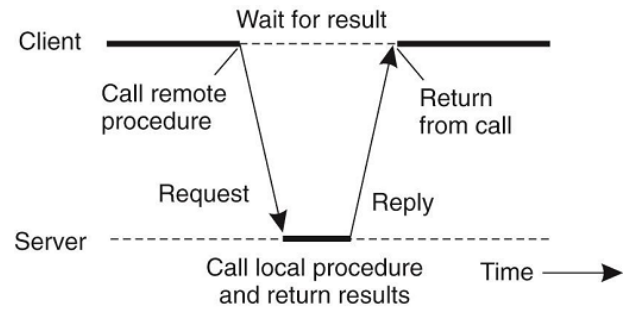
- **Socket programming: thread models (patterns) for handling connections from multiple clients**
  - **Thread per connection**
    - Server creates a new thread every time a new client connects.
  - **Thread per request**
    - Server creates a new thread every time a new request is made (one connection can include several requests)
  - **Delegation (Manager-Workers, thread per socket)**
    - Server (`manager`) accepts all clients at the same welcoming socket, then delegates the communication with each request from a client to a `worker` function `handle_client(conn)` that runs in a separate thread and uses a different socket.
  - **Producer-Consumer pattern (pre-fork)**
    - The server (`producer`) maintains a thread-safe queue of client connections.
    - A pre-defined set of threads is created once the application started (not on demand like previous methods).
    - The worker functions (`consumers`) running in separate threads (chosen from the pool of available threads at that time)
    - Number of clients server simultaneously =  $\min(\text{queue\_size}, \text{thread\_pool\_size})$
  - **Combined pipeline:**
    - Several patterns can be combined if tasks (e.g., receiving, processing, sending) are separated.
    - We can assign workers to groups based on their tasks; each group `may` have a manager thread, each worker will have an interface (take tasks, process them, and pass to next worker).



## Lecture 5-6: RPC and System Architecture

### • Remote Procedure Call (RPC)

- Allows programmer to call procedures located on other machines.
- Procedures should be isolated (work in a black box) and to have one specific job.
- A **stub** code is used in client to build message, receive and unpack results, and in server to receive, unpack, and deliver message to the server, and then pack results and return them to client.
- Packaging the message means encoding it to a sequence of bytes (to be sent through the network), thus, client and server has to agree on the same encoding method.
- Asynchronous RPC allows the client to continue doing its work without busy-waiting for the server response.
- Client can utilize multi-casting to send multiple RPC requests to group of servers.
- **xmlrpc** is a standard library in python that implements RPC.



**Stub** is a piece of code used to stand in for some functionality.

A stub may simulate the behavior of existing code (e.g., a procedure on a remote machine) or be a placeholder for yet-to-be-developed code.

### • System Architecture:

#### ○ Centralized Architecture

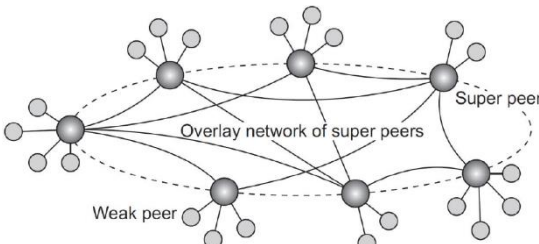
- **Single-tiered:** the whole application (frontend and backend) is running on some server (mainframe) and users can use it by connecting to it (either physically or through a terminal).
- **Two-tiered:** one server offers all services, clients running some frontend.
- **Three (or more)-tiered:** the system is divided into several logical layers, each doing one or more jobs.
  - **Example:** web server acting as an entry point for requests, it send data to the processing server which communicates with another database server.

Vertical distribution	Horizontal distribution
Each logical layer (e.g., UI, Application, Data) lies in a separate machine.	Distributing a single logical layer (e.g., a database server) on several machines.

Fat client	Thin client
Implements many logical layers at the client-side. Fat clients are hard to manage as more effort is needed to handle any user behavior, and multiple versions are needed to support many platforms.  <b>Example:</b> a client that runs UI, Application, and possibly a local DB.	Implements only the necessary parts at the client-side, the rest of the system is implemented at server(s). Thin clients are preferable and easier to manage.  <b>Example:</b> a client that run all (or part of) UI, and possibly a part of the application.

## ○ Decentralized (P2P) Architecture

- Each peer (node) acts as a server and client simultaneously.
- System functionality is **horizontally distributed** over **equally-privileged nodes**.
- **Overlay network**: a representation/topology of some network (e.g., p2p) that is implemented on top of the real unstructured network.
  - A link in overlay network corresponds to a real path of possibly many physical links.

Structured P2P Network	Unstructured P2P Network
<ul style="list-style-type: none"> <li>- Overlay network is structured as nodes and some direct links between them.</li> <li>- Each node has an id and is responsible for holding the values of one or more data items.</li> <li>- <b>To lookup a data item</b>, we typically use a <u>Distributed Hash Table (DHT)</u> in which <u>hashes</u> (of data items' values or randomly-generated item ids) <u>are used as keys</u> to determine their location.</li> <li>- One or more keys are then mapped to a node id using some mechanism that should be efficient and deterministic.</li> <li>- DHT implementations include: <a href="#">Chord</a> (*), <a href="#">Pastry</a>, <a href="#">Tapestry</a>, <a href="#">Kademlia</a>, <a href="#">Bamboo</a></li> </ul>	<ul style="list-style-type: none"> <li>- Each node maintains an updated list of neighbors.</li> <li>- The result is a random graph (an edge between nodes <math>u, v</math> exists with a certain probability). Ideally this probability has to be the same for any <math>u, v</math>.</li> <li>- <b>To lookup a data item</b>, we need to perform a search algorithm (e.g., flooding (**), random walk (***)).</li> <li>- <b>Scalability problem</b>: as the network grows, locating data items becomes inefficient.</li> <li>- <b>One solution</b> can be to use special nodes (super-peers) that maintains an index of data items.</li> </ul> 

### (\*) Chord DHT

- Nodes are logically organized in a ring; each node has an  $m$  bit identifier and maintains a finger table.
- Finger table stores shortcut links between nodes that help making search faster ( $\log N$ )
- Each data item is hashed to an  $m$  bit key. Data item with key  $k$  is stored at node with smallest id  $\geq k$ , called the successor of key  $k$ .
- When a new node joins the chord, node successors, key placement, and finger tables are updated.

### (\*\*) Flooding

- When any peer is queried about the location of a data item:
  - It searches locally for the data. If found, it returns to the caller which propagates it back.
  - Otherwise, it floods (i.e., forward the request to all its neighbors).
- If a node receives the same request multiple times, it ignores the request.
- A TTL limit can be introduced that is pre-chosen, or sender can start with TTL=1 then receivers increase TTL if data was not found.

### (\*\*\*) Random walks

- Issuing node passes request for data item to a randomly chosen neighbor.
- **Pros**: less bandwidth than flooding
- **Cons**: can be very slow (can be improved by starting multiple random walks simultaneously)
- Let  $N$ : number of nodes,  $r$  number of replicas.
- Probability that item is found after  $k$  attempts:
 
$$P(k) = \frac{r}{N} \left(1 - \frac{r}{N}\right)^{k-1}$$
- A TTL should also be introduced to limit the search size  $S = \frac{N-1}{r}$
- Random walks are more communication efficient (on average) but might take longer.

- **Hybrid Architecture**

- **Edge-server systems:**

- **Edge servers:** servers that are dedicated for end-users to contact for service (can be ISP servers, or content delivery servers distributed by some company).
    - They operate at the **edge** between the enterprise network and the public internet.
    - They may share data between each other in a p2p manner (hybrid architecture).

- **Collaborative distributed system**

- A server is maintained to allow peers to join the network and get started
    - Once peers join, they don't need the server (decentralized collaboration).
    - **Example:** [BitTorrent](#) protocol for p2p file sharing.
      - Torrent file is hosted by some server, it contains a reference to tracker server.
      - Tracker server maintains a list (swarm) of nodes (**peers** or seeds) that have (**chunks of**) the file.

## Lecture 7-8: Coordination

- **Coordination**

- Manages the interaction and dependencies between activities in a distributed system.
- Coordination encapsulates synchronization.

- **Synchronization**

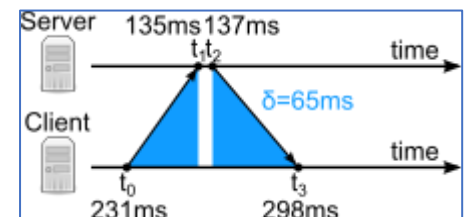
- Ensuring that all nodes in a distributed system agree on a timing mechanism (clock time or relative ordering of events) so they can collaborate and not conflict.

Process synchronization	Data synchronization	Clock synchronization
Ensuring that processes agree on the sequence of actions to be taken.	Ensuring that data in two nodes stays the same.	Ensure that independent clocks in nodes are synchronized.

Physical clocks (*)	Logical clocks
Maintains the current Universal Coordinated Time (UTC) broadcasted through a short-wave radio and satellite ( $\pm 0.5\text{ms}$ accuracy)	Maintains a counter (or similar concept that represents time) and use it to ensure a proper ordering of events.
<ul style="list-style-type: none"> <li>- Network Time Protocol</li> <li>- Berkley Algorithm</li> </ul>	<ul style="list-style-type: none"> <li>- Lamport's logical clocks</li> </ul>
<b>Clock Precision (<math>\pi</math>):</b> <ul style="list-style-type: none"> <li>- Difference between clock times calculated by two nodes (synchronized internally between processes)</li> </ul> <b>Clock Accuracy (<math>\alpha</math>):</b> <ul style="list-style-type: none"> <li>- Difference between clock time calculated by a node and UTC (synchronized using external server).</li> </ul>	

- **Network Time Protocol (NTP)**

- NTP client polls server for time at  $t_0$ , server gets poll at  $t_1$ , server sends answer at  $t_2$ , answer reaches client at  $t_4$
- $t_0, t_3$  are measured in client time system.  $t_1, t_2$  in server's.
- The client can now calculate two values:
  - RTT ( $\delta$ ): response time – processing time.
  - Time offset ( $\theta$ ): the time difference between client and server (textbook omits the abs operator).
- The client can collect  $N$  pairs ( $\delta, \theta$ ) from  $N$  NTP servers and chooses the  $\theta$  for which  $\delta$  was minimal.
- Using the knowledge of  $\theta$ , client adjusts the rate of its clock to minimize offset.



$$\delta = (t_3 - t_0) - (t_2 - t_1)$$

$$\theta = \left| \frac{(t_1 - t_0) + (t_2 - t_3)}{2} \right|$$

- **Berkley Algorithm**

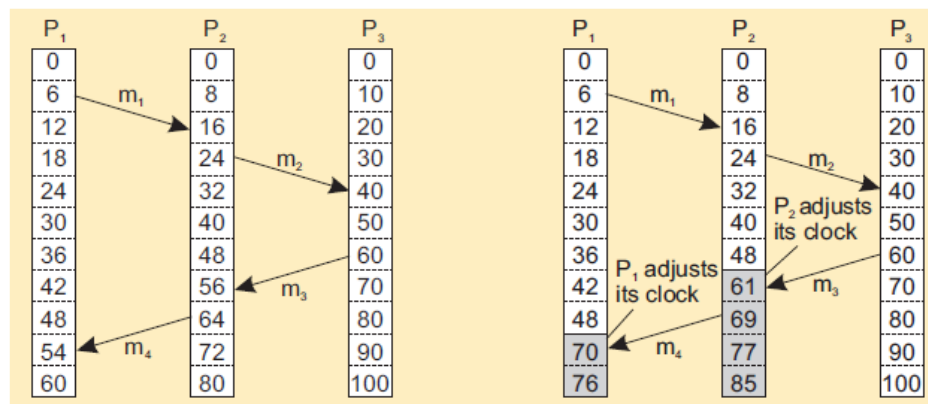
- A **time server** scans all machines periodically and tells each one how it should adjust its rate (go faster or go slower).

- **Lamport's Logical Clocks** (*constructing totally-ordered set of events to agree on event precedence*)

- **Idea**

- Each process  $P_i$  maintains a local counter  $C$  (represents timestamp), increments when events occur (counters are handled in the middleware level).

- Each event (message)  $e_i$  has a timestamp, where  $C(e_1) < C(e_2)$  means that  $e_1$  **happened before**  $e_2$
- Whenever  $P_i$  sends a message to  $P_j$ , it attaches the current  $C_i$ , then  $P_j$  updates  $C_j = \max(C_j, C_i)$  before passing message to the application.
- Use Process ID as tie-breaker (if  $C_i = C_j$ ) then the process with lower PID is assumed to execute first)
- **Implementation (total ordered multicast)**
  - Whenever  $P_i$  wants to send a message to  $P_j$ , it sends a timestamped acknowledgement to all processes (including itself).
  - Each process maintains a queue of timestamped messages it received, a process  $P_j$  only passes message  $m$  to the application if  $m$  is the head of the queue and all the requests received by  $P_j$  (from all other processes, assuming reliability) have larger timestamps.



- **Mutual Exclusion Algorithms:** coordinate access to a shared resource by allowing only one process to use it at a time.

	Centralized, permission-based	Distributed, permission-based
<b>Desc.</b>	Processes require coordinator permission whenever they access the resource.	A process accessing the resource multicasts the message (resource_name, pid, current_time) <b>If the receiving process:</b> <ul style="list-style-type: none"> <li>• Currently using resource → send nothing, queue request</li> <li>• Doesn't want to use the resource → send OK.</li> <li>• Also want to use the resource → compare timestamps, the earlier request wins (sends OK if the sender wins. sends nothing, queues the request otherwise).</li> </ul>
<b>Pros</b>	<ul style="list-style-type: none"> <li>• Guaranteed correctness</li> <li>• Fair (first to request gets)</li> <li>• Simple, No starvation.</li> </ul>	<ul style="list-style-type: none"> <li>• Guaranteed correctness</li> <li>• No deadlock</li> <li>• No starvation.</li> </ul>
<b>Cons</b>	<ul style="list-style-type: none"> <li>• Single point of failure</li> <li>• Silent failure of coordinator.</li> <li>• Performance bottleneck.</li> </ul>	<ul style="list-style-type: none"> <li>• <b>Congesting network:</b> <math>2 * (N - 1)</math> messages per resource acquisition</li> <li>• <b>N points of failure:</b> if one process crashes, others will wait forever.</li> <li>• <b>Inefficiency:</b> each process maintains an updated list of all processes using the resource, all processes are involved in all resource acquisition decisions.</li> </ul>



- **Token Ring (distributed, token-based algorithm)**
  - Processes are organized in a logical ring; a token is passed between them.
  - The one which has the token may proceed in its critical section, or pass if not interested.
  - **Pros:** simple, guaranteed correctness, no starvation.
  - **Cons:**
    - Token can get lost (holder crashes, passing message was lost).
    - Ring needs to be maintained (detect crashes and enroll/unenroll processes).
- **Decentralized, voting-based algorithm**
  - Resource is assumed to be replicated (with its coordinator) on  $N$  hosts.
  - Coordinator should allow only one access to the resource, it can crash (forgets given permissions) but should recover quickly.
  - A process acquiring a resource holds an election, the process with majority votes ( $> \frac{N}{2}$ ) wins.
    - Losers backoff and retry after a random period of time.
  - **Probability of correctness violation**  $= \sum_{k=2m-N}^m \binom{m}{k} p^k (1-p)^{m-k}$  is negligible.
    - **Correctness violation:** most of coordinators who permitted resource acquisition crashed during the same time interval.
    - $p$ : average availability of coordinators.
    - $m$ : number of coordinators voting **for** the resource acquisition.
      - Best case:  $m = N$ , worst case  $m = \frac{N}{2} + 1$
  - **Pros:** decentralized, fault tolerant (up to  $2m - N$  simultaneous crashes are tolerated)
  - **Cons:** utilization drop (if many nodes want the resource, none will get enough votes and they will keep random-wait-retry until success).
- **Election Algorithms:** used to select a special node **S** (e.g., a coordinator, leader, super-peer) in a distributed system.
  - **The bully algorithm:**
    - Assign ids to processes, the available process with the highest *id* is **S**.
    - Whenever  $P_i$  detects that **S** is down, it sends ELECTION to all processes with higher-ids.
      - If any replied, it becomes **S** and informs all others.
      - Otherwise,  $P_i$  becomes **S**.
    - Whenever a process recovers, it holds an election.
  - **The ring algorithm**
    - Processes are organized in a logical ring, with each one knowing its successor.
    - Whenever  $P_i$  detects that **S** is down, it sends  $L = [i]$  to its successor  $P_j$
    - $P_j$  append  $[j]$  to  $L$  and forwards to its successor.
    - When  $L$  reaches  $P_i$ , election happens:  $S = \max(L)$  and all processes are informed.
  - **Election in wireless environments**
    - // Check the textbook.



## Lecture 9-10: Consistency and Replication

- **Replication in distributed systems:**

- Storing copies of data across nodes to **increase performance** (data is closer to consumers, load is distributed) **and reliability** (if one node is down, system still function) of the system.
- **Replication issues:**
  - Where to place replicas (everywhere → slow system, big bandwidth consumption)
  - How to place replicas (what algorithm and mechanisms to use?)
  - By whom (which location has the up-to-date data that should be replicated)
  - The above parameters determine how expensive is it to update replicas.
- **Replica management** is concerned about solving the above issues.
- **Replica placement**
  - Where to place  $N$  servers of the distributed system.
  - Where to place  $K$  replicas of data across them such that the average distance to all clients is minimal.
- **Replica types**

Permanent replicas	Server-initiated replicas	Client-initiated replicas
Few amounts of data (core system) distributed around a limited number of servers (e.g., website files replicated across mirror sites)	Created by data store owner to enhance the performance of the system.	Created by users updating their data.
Often useful as a backup.	Often used for placing read-only copies closer to clients.	More like a client cache.

- **Update propagation:** what to propagate?

	Update notification	Updated data (Passive/Push-based)	Update operation (Active/Pull-based)
<b>Bandwidth usage</b>	Low	High	Efficient
<b>When to use</b>	When updates are more frequent than reads	When reads are more frequent than updates	When update operation is not complex.
<b>Approach</b>	Multicast	Multicast	Unicast

// Leases: age-based, renewal-frequency based, state-based.

- **Consistency Model (CM)**

- A contract (set of rules) between processes and a (distributed) data store they communicate with.
- The store will guarantee consistent data as long as processes follow the contract.
- **Read about the following models.**

Data-Centric CM	Client-Centric CM
Continuous consistency Consistent ordering of operations Eventual consistency	Monotonic reads Monotonic writes Read your writes Writes follows reads

- **Consistency Protocol:** describes the implementation of a specific consistency model
  - **Primary-based protocols:** *write once in primary location, read everywhere.*
    - **Primary backup protocol:**
      - All writes are forwarded to the primary server for updating and propagating changes.
      - Primary server acks all changes only after read-only copies are available in all replicas.
      - **Pros:** writes are globally ordered, system is highly fault-tolerant.
      - **Cons:** update is a blocking operation (may take a lot of time to finish)
    - **Variant 1, non-blocking:** flip the order (primary server acks first, then propagate updates).
      - Makes system faster but less fault-tolerant.
    - **Variant 2, allows local writes:** when writing to  $x$  at server  $i$ , that server becomes the new primary location for  $x$  and informs/updates all replicas.
      - Efficient for successive writes to same location, with reads in different locations.
  - **Replicated-write protocols:** *write in multiple replicas, read everywhere.*
    - **Active replication:** uses a global coordinator (sequencer) that assigns a Sequence Number (SN) to each update to guarantee that writes are ordered.
    - **Quorum-based replication:** reads/writes require majority voting from servers, all replicas are assigned a version number.
  - **Cache-coherence protocols**
    - Used to maintain consistency between data stored in multiple local caches across users.