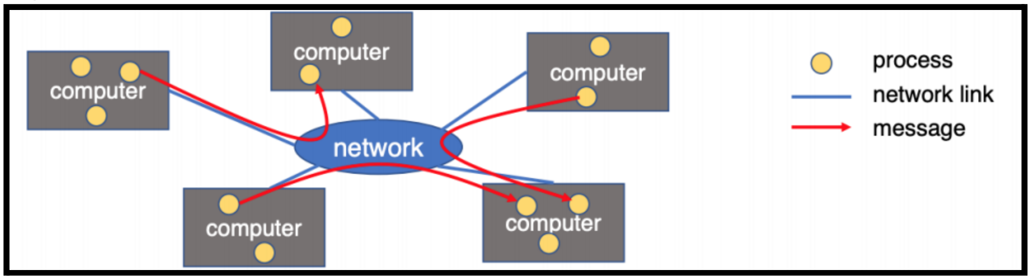
Lecture Notes:

- **Distributed Systems:**
- A **distributed system** is cooperating processes in a computer network.
- It is a group of computers working together as to appear as a single computer to the end-user. These machines have a shared state, operate concurrently and can fail independently without affecting the whole system's uptime.
- E.g.



- Some popular distributed systems today include:
    - Google file systems
    - BigTable
    - MapReduce
    - Hadoop
    - ZooKeeper
- There are 3 degrees of integration for distributed systems:
    1. **Loosely-coupled:** E.g. internet applications (email, web, FTP, SSH).
    2. **Mediumly-coupled:** E.g. remote execution (RPC), remote file system (NFS).
    3. **Tightly-coupled distributed:** E.g. file systems (AFS)
- Advantages of distributed systems:
    1. Performance - parallelism across multiple nodes.
    2. Scalability - by adding more nodes.
    3. Reliability - leverage redundancy to provide fault tolerance.
    4. Cost - cheaper and easier to build lots of simple computers.
    5. Control - users can have complete control over some components.
    6. Collaboration - much easier for users to collaborate through network resources.
- The promise of distributed systems:
    1. Higher availability - when one machine goes down, use another.
    2. Better durability - store data in multiple locations.
    3. More security - each piece is easy to secure.
- The reality of distributed systems:
    1. Worse availability - depend on every machine being up.
    2. Worse reliability - can lose data if any machine crashes.
    3. Worse security - anyone in the world can break into the system.

  Coordination is more difficult - must coordinate multiple copies of shared state information (using only a network).

- Requirements:
    - **Transparency:** The ability of the system to mask its complexity behind a simple interface.
    - Possible transparencies:
        - Location - cannot tell where resources are located.
        - Migration - resources may move without the user knowing.
        - Replication - cannot tell how many copies of resources exist.
        - Concurrency - cannot tell how many users there are.
        - Parallelism - may speed up large jobs by splitting them into smaller pieces.
        - Fault Tolerance - system may hide various things that go wrong.
    - Transparency and collaboration require some way for different processors to communicate with one another.
- Clients and Servers:
    - The prevalent model for structuring distributed computation is the client/server paradigm.
    - A **server** is a program or collection of programs)that provides a service.
    - The server may exist on one or more nodes.
    - **Note:** Often the node is called the server, too, which is confusing.
    - A **client** is a program that uses the service.
    - A client first binds to the server (locates it and establishes a connection to it). Then, the client sends requests, with data, to perform actions, and the server sends responses, also with data.
- Naming:
    - Essential naming systems in network:
    - Address processes/ports within the system (host, id) pair.
    - Physical network address (Ethernet address).
    - Network address (Internet IP address).
    - Domain Name Service (DNS) provides resolution of canonical names to network addresses.
- Communication:
    - There are a few ways computers can communicate with each other:
    1. **Raw Message - UDP:**
        - Network programming = raw messaging (socket I/O).
        - Programmers hand-coded messages to send requests and responses.
        - This method is too low-level and tiresome.
        - Need to worry about message formats.
        - Must wrap up information into a message at source.
        - Must decide what to do with the message at the destination.
        - Have to pack and unpack data from messages.
        - May need to sit and wait for multiple messages to arrive.
    2. **Reliable Message - TCP:**
    3. **Remote Procedure Call (RPC) and Remote Method Invocation(RMI):**
        - **Procedure calls** are a more natural way to communicate.
        - Every language supports them.
        - Semantics are well-defined and understood.
        - Natural for programmers to use.
        - The idea is to let servers export procedures that can be called by client programs.
        - Similar to module interfaces, class definitions, etc.

- Clients just do a procedure call as if they were directly linked with the server.
- Under the covers, the procedure call is converted into a message exchange with the server.
- **Remote Procedure Call (RPC)** is the most common means for remote communication.
- It is used both by operating systems and applications.
- DCOM, CORBA, Java RMI, etc., are all basically just RPC. NFS is implemented as a set of RPCs.
- A server defines the server's interface using an **Interface Definition Language (IDL)** that specifies the names, parameters, and types for all client-callable server procedures.
- A **stub compiler** reads the IDL and produces two stub procedures for each server procedure (client and server).
- Server programmer implements the server procedures and links them with server-side stubs.
- Client programmer implements the client program and links it with client-side stubs.
- The **stubs** are the "glues" responsible for managing all details of the remote communication between client and server. They send messages to each other to make RPC happen transparently.
  A client-side stub packs the message, sends it off, waits for the result, unpacks the result and returns to the caller.
  A server-side stub unpacks the message, calls the procedure, packs the results, sends them off.
- **Marshalling** is the packing of procedure parameters into a message packet.
- The RPC stubs call type-specific procedures to marshal or unmarshal the parameters to a call.
  The client stub marshals the parameters into a message.
  The server stub unmarshals parameters from the message and uses them to call the server procedure.
- On return:
    - The server stub marshals the return parameters.
    - The client stub unmarshals return parameters and returns them to the client program.