# Module 4: Communication

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

# Contents

- Introduction to Distributed Systems
- Characterization of Distributed Systems
- Distributed Shared Memory Approaches
- Message Passing
- Programming Using the Message-Passing Paradigm
  - Group Communication
- Case Study (RPC and Java RMI)

# Introduction to Distributed Systems

# Examples of distributed systems

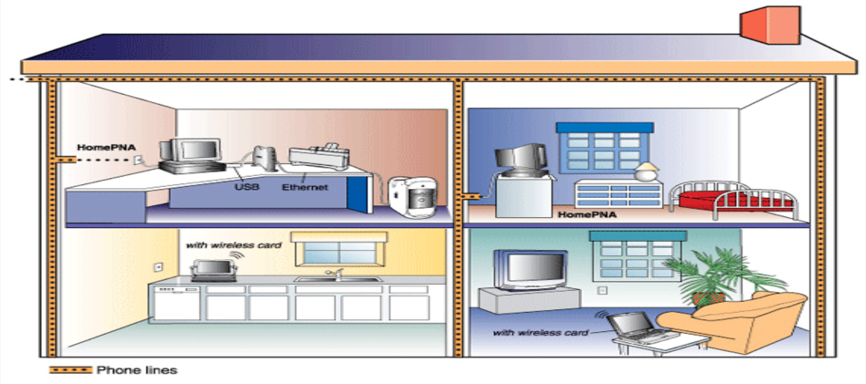
- NFS
- The internet
- DNS
- Amazon AWS
- Google Cloud Platform

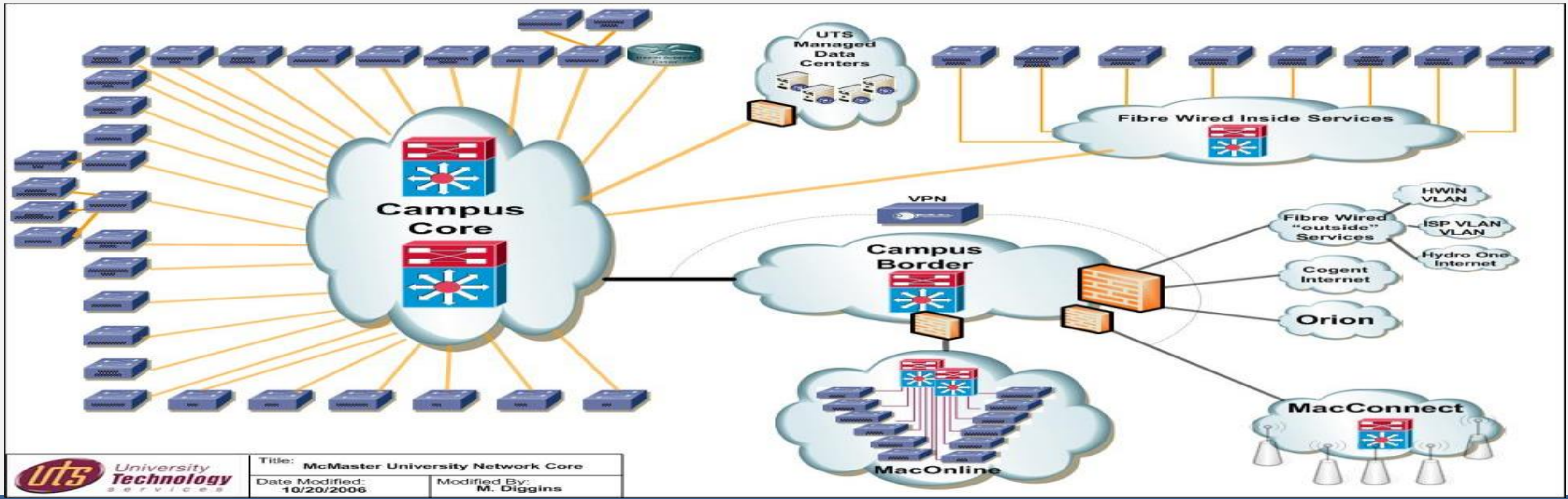
# Introduction

- Networks of computers are everywhere!
  - Mobile phone networks
  - Corporate networks
  - Factory networks
  - Campus networks
  - In-car networks
  - **Internet of Things (IoT)**
  - On board networks in planes and trains

- Mobile - GSM/3G networks, cell towers connect to backbones
- Corporate - LAN, consisting of PC’s, laptops, servers, storage, printing
- Factory - Machines (drills, stamps, robots) communicate over network
- Campus - Like CSSE Dept / Unimelb. Wired/wireless, clients, servers
- Home - Laptops, PC’s, Xbox/Playstation, Set top boxes
- In-car - Network of sensors (monitoring engine, breaks, tires etc)

# Characteristics of networked/distributed computing systems and applications



# What is Distributed System?

- Collection of entities (along with some memory) to work towards a common goal
- Distribution is Transparent to the users and appears as one local machine

*A distributed system as one in which hardware or software components located at networked computers communicate and coordinate their actions only by message passing*.-[Coulouris]

*A distributed system is a collection of independent computers that appear to the users of the system as a single computer*.-[Tanenbaum]

# Generic Components of Distributed System

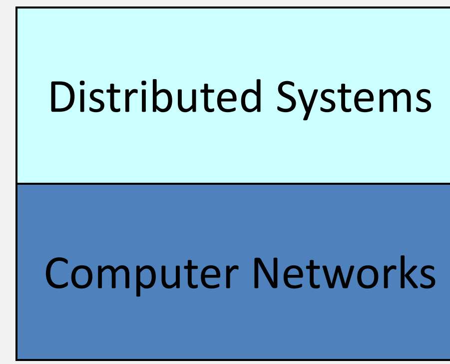
- Computing system
  - Mostly heterogeneous
  - Multicore, general purpose / embedded processing, SIMD or MIMD
- Communication
  - LAN, Internet

# How is distributed system different from Networked system?

- In networked systems, user is aware of individual node, their location and functionality
- Distributed system appears as a single machine to the users

# Networks Vs Distributed Systems

- Networks: A media for interconnecting local and wide area computers and exchange messages based on protocols. Network entities are visible and they are explicitly addressed (IP address).
- Distributed System: existence of multiple autonomous computers is transparent
- However,
  - many problems (e.g., openness, reliability) in common, but at different levels.
    - Networks focuses on packets, routing, etc., whereas distributed systems focus on applications.
    - Every distributed system relies on services provided by a computer network

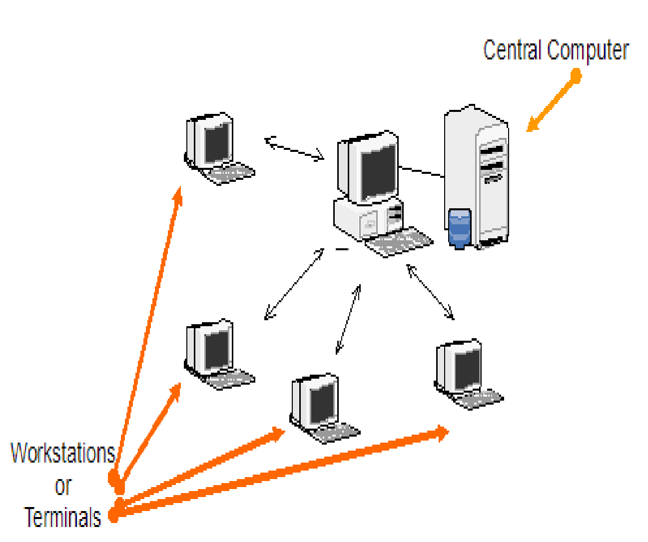


# Reasons for Distributed Systems

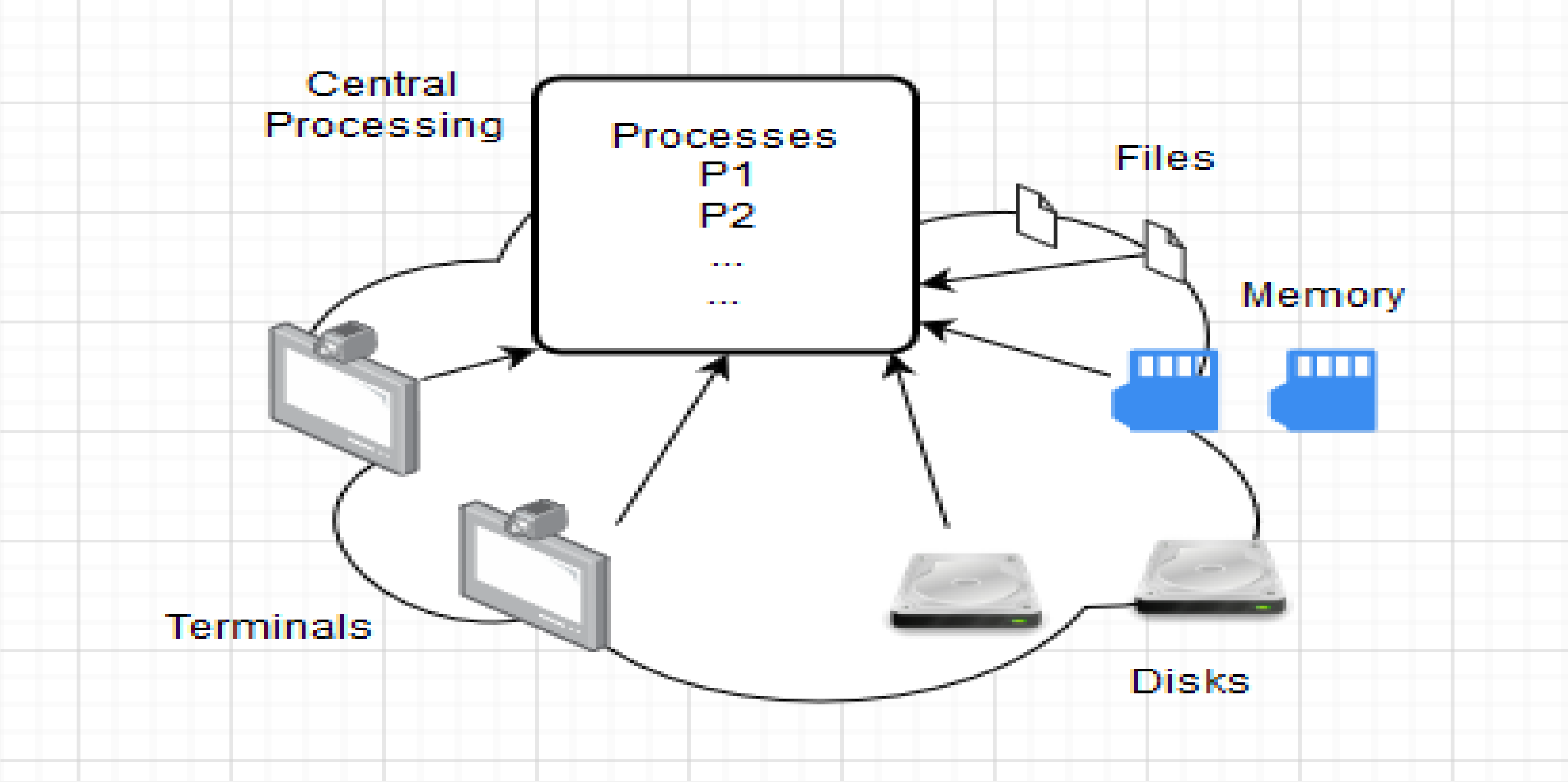
- Functional Separation:
  - Existence of computers with different capabilities and purposes:
    - Clients and Servers
    - Data collection and data processing
- Inherent distribution:
  - Information:
    - Different information is created and maintained by different people (e.g., Web pages)
  - People
    - Computer supported collaborative work (virtual teams, engineering, virtual surgery)
  - Retail store and inventory systems for supermarket chains (e.g., Coles, Woolworths)

# Reasons for Distributed Systems

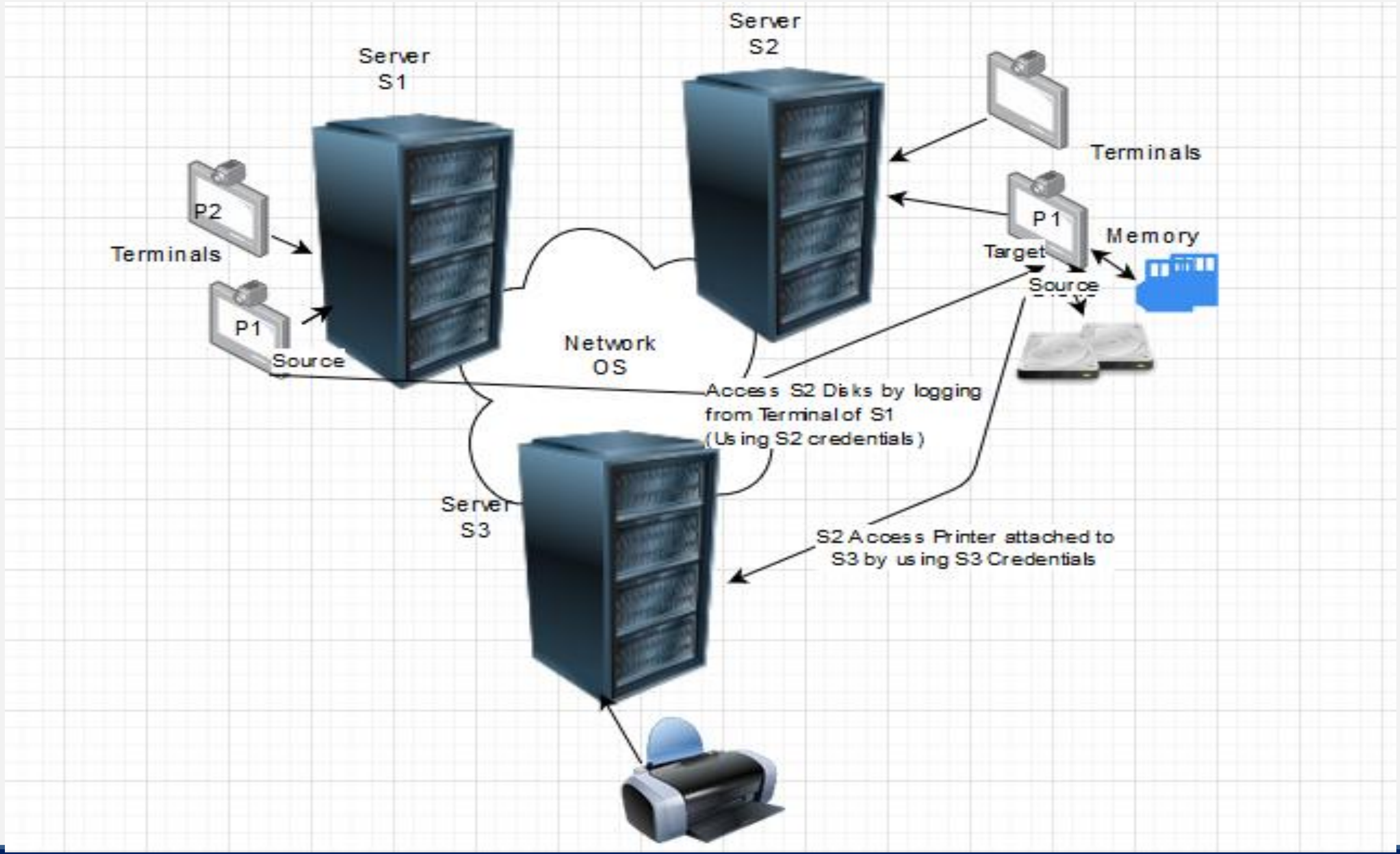
- Power imbalance and load variation:
  - Distribute computational load among different computers.
- Reliability:
  - Long term preservation and data backup (replication) at different locations.
- Economies:
  - Sharing a printer by many users and reduce the cost of ownership.
  - Building a supercomputer out of a network of computers.



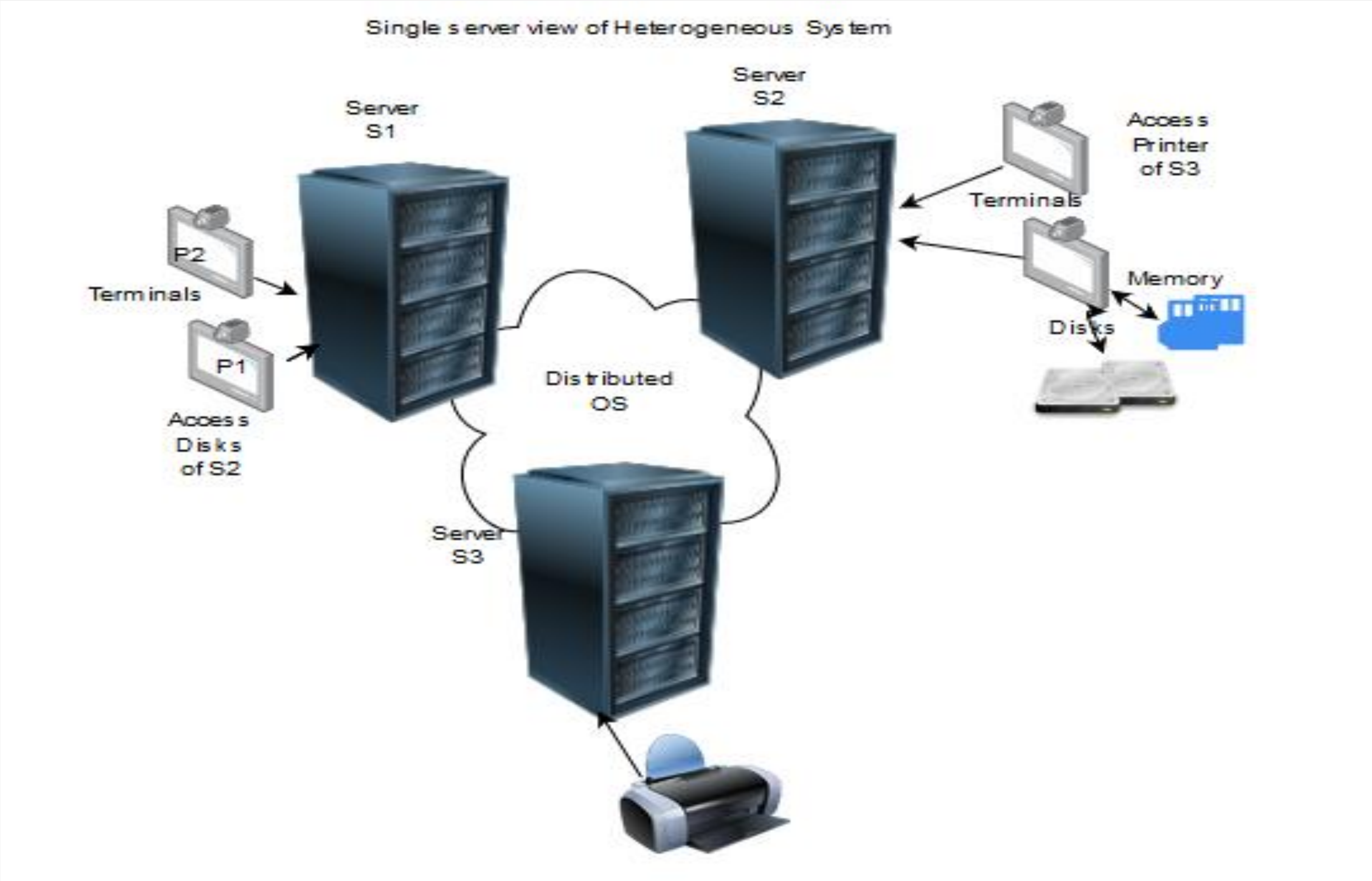
# Centralized System



# Networked System



# Distributed System



# Consequences of Distributed Systems

- Computers in distributed systems may be on separate continents, in the same building, or the same room. DSs have the following consequences:
  - Concurrency – each system is autonomous.
    - Carry out tasks independently
    - Tasks coordinate their actions by exchanging messages.
  - Heterogeneity
  - No global clock
  - Independent Failures

# Characterization of Distributed Systems

# Significant Characteristics

- Parallel activities
  - Autonomous components executing concurrent tasks
- Communication via message passing
  - No shared memory
- Resource sharing
  - Printer, database, other services
- No global state
  - No single process can have knowledge of the current global state of the system
- No global clock
  - Only limited precision for processes to synchronize their clocks

# Goals of Distributed Systems

- Connecting Users and Resources
- Transparency
- Openness
- Scalability
- Enhanced Availability

- Distributed computing as a utility
  - Cluster computing
  - Grid computing
  - Cloud computing

# Typical Distributed System Design Goals

- Heterogeneity
- Availability
- Scalability
- Transparency
- Concurrency
- Efficiency
- Robustness

# Other Ideal Characteristics

- Efficient Resource Management: Middle ware should manage efficiently heterogeneous systems
- Effective Synchronization:  Issues in the absence of global clock should be effectively managed
- Security
- Complexity
- Highly variable bandwidth and latency
- Openness – Distributed system should be extensible
- Transparency
- Quality of Service

# Examples of Distributed Systems

- They (DS) are based on familiar and widely used computer networks:
  - Internet
  - Intranets, and
  - Wireless networks

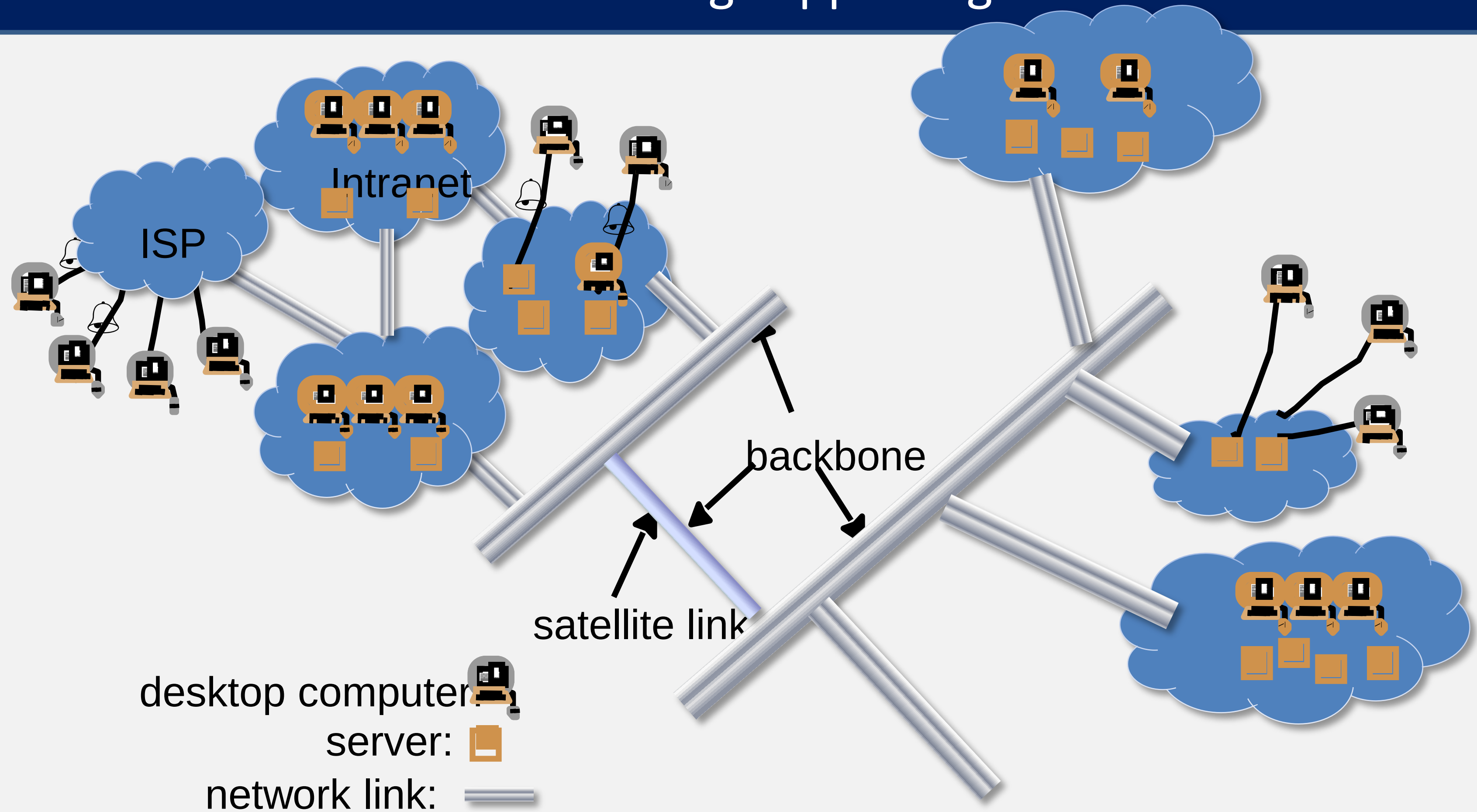

- Example DS and its Applications:
  - Web (and many of its applications like Online bookshop)
  - Data Centers and Clouds
  - Wide area storage systems
  - Banking Systems
  - User-level communication (Facebook, Zoom)


# Selected application domains and associated networked applications

| | |
|---|---|
| *Finance and Commerce* | eCommerce e.g. Amazon and eBay, PayPal, online banking and trading |
| *The information Society* | Web information and search engines, ebooks, Wikipedia; social networking: Facebook and Twitter. |
| *Creative Industries and Entertainment* | Online gaming, music and film in the home, user-generated content, e.g. YouTube, Flickr |
| *Healthcare* | Health informatics, on online patient records, monitoring patients (Metro South Health hospital trial in Queensland) |
| *Education* | e-learning, virtual learning environments; distance learning. e.g., Coursera |
| *Transport and Logistics* | GPS in route finding systems, map services: Google Maps, Google Earth |
| *Science and Engineering* | Cloud computing as an enabling technology for collaboration between scientists (LHC, LIGO) |
| *Environmental Management* | Sensor networks to monitor earthquakes, floods or tsunamis (Bureau of Meteorology flood warning system) |

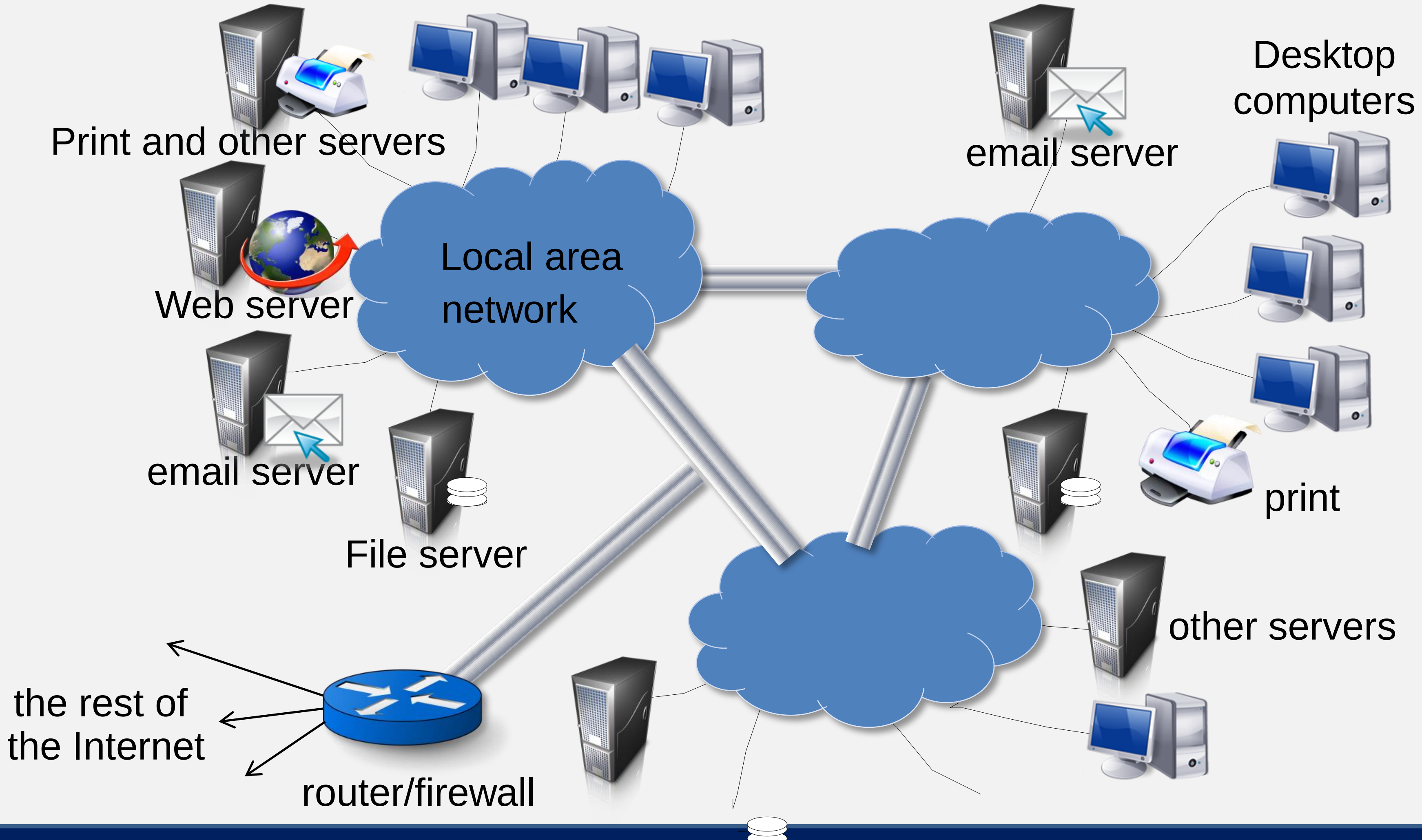
# A typical portion of the Internet and its services:

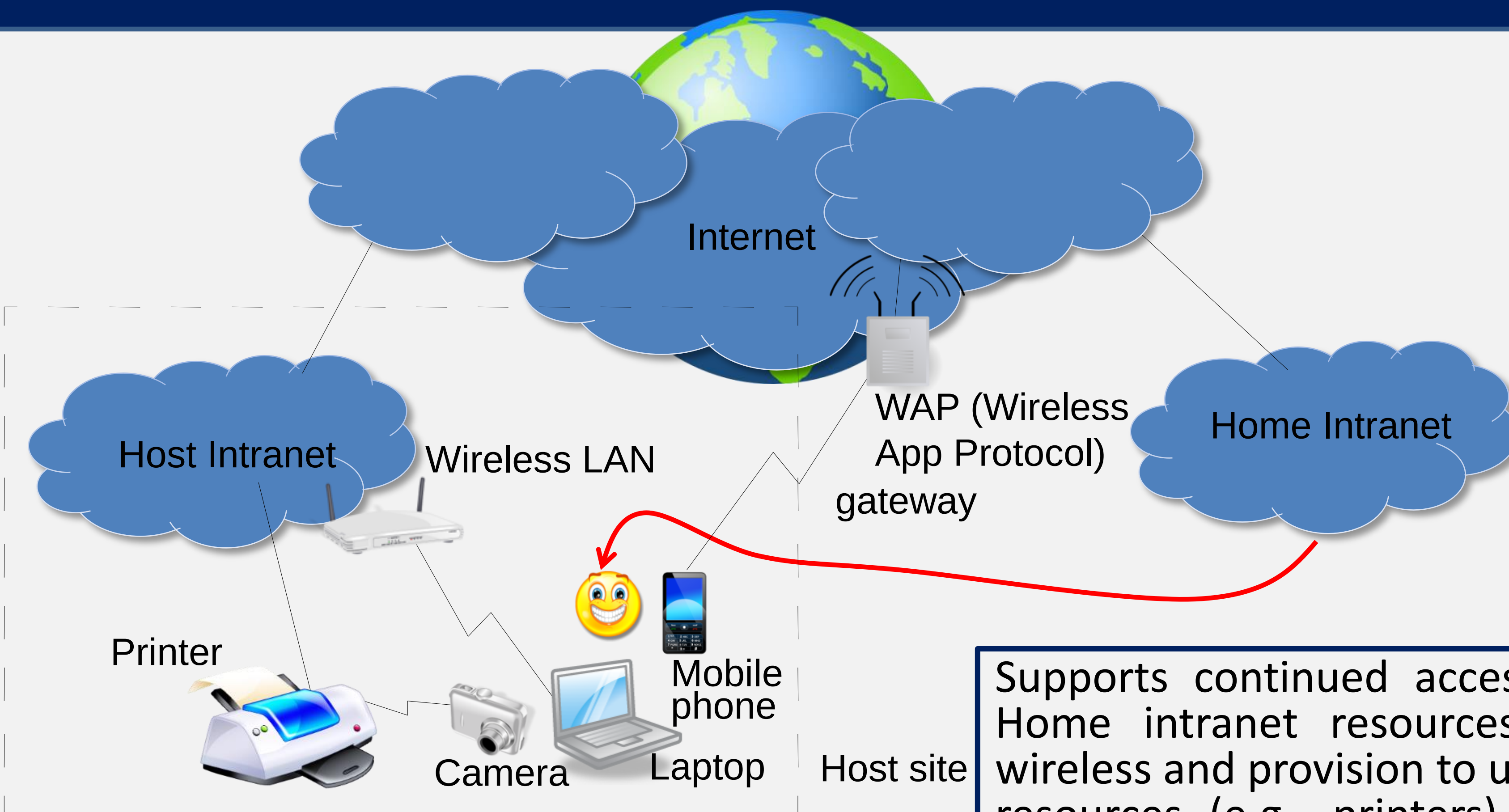
Multimedia services providing access to music, radio, TV channels, and video conferencing supporting several users



# A typical Intranet

A portion of Internet that is separately administered & supports internal sharing of resources (file/storage systems and printers)

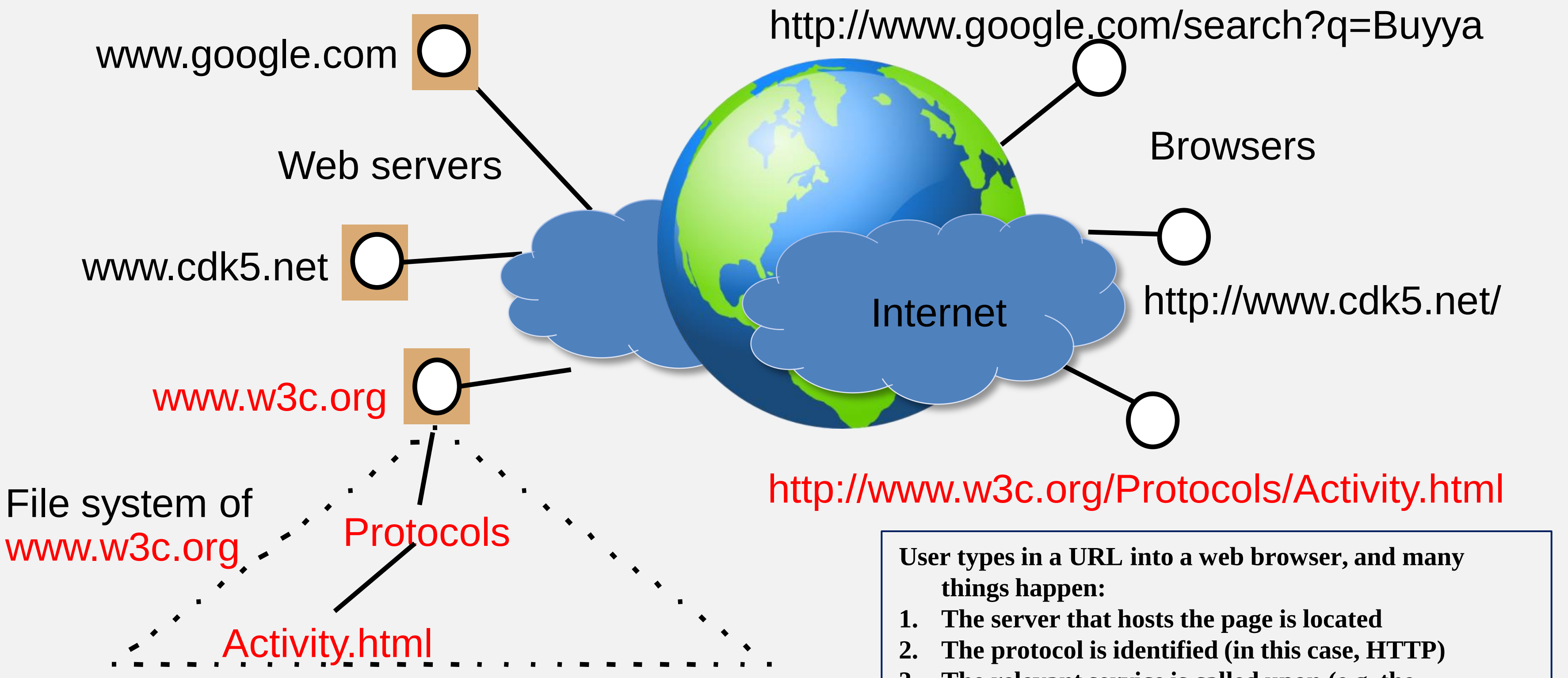





Supports continued access to Home intranet resources via wireless and provision to utilise resources (e.g., printers) that are conveniently located (location-aware computing).

- mobile computing
- location/contextaware computing
- ubiquitous computing
- spontaneous interoperation
- service discovery

# Resource sharing and the Web: open protocols, scalable servers, and pluggable browsers




**User types in a URL into a web browser, and many things happen:**

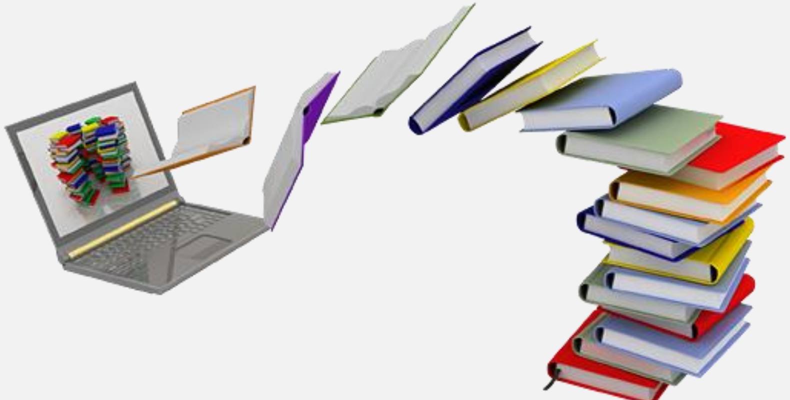
1. **The server that hosts the page is located**
2. **The protocol is identified (in this case, HTTP)**
3. **The relevant service is called upon (e.g. the webserver) to satsify request**
4. **The particular page that is requested is located and returned to user**

# Trends in Distributed System

- Emergence of pervasive technologies – almost always connected
- Increased Multimedia support and use
- Distributed systems as utility → Cluster, Cloud, Grid

# Business Example and Challenges

- Online bookstore (e.g. in World Wide Web)
  - Customers can connect their computer to your computer (web server):
    - Browse your inventory
    - Place orders
    - …



This example has been adapted from **Torbin Weis**, Berlin University of Technology

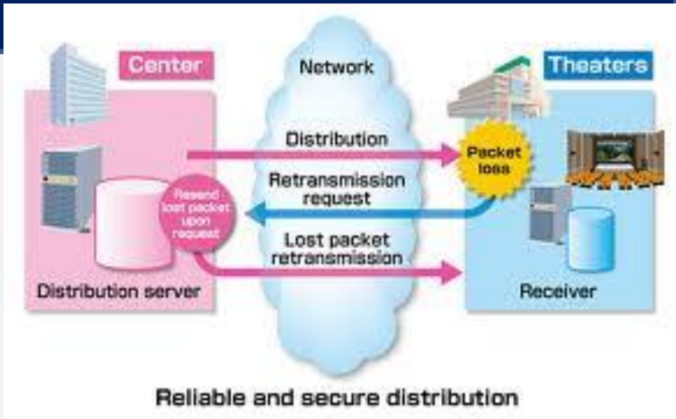
# Business Example – Challenges I

- What if
  - Your customer uses a completely different hardware? (PC, MAC, iPad, Mobile…)
  - … a different operating system? (Windows, Unix,…)
  - … a different way of representing data? (ASCII, EBCDIC,…)
  - **Heterogeneity**
- Or
  - You want to move your business and computers to the Caribbean (because of the weather or low tax)?
  - Your client moves to the Caribbean (more likely)?
  - **Distribution transparency**

# Business Example – Challenges II

- What if
  - Two customers want to order the same item at the same time?
  - **Concurrency**
- Or
  - The database with your inventory information crashes?
  - Your customer's computer crashes in the middle of an order?
  - **Fault tolerance**

# Business Example – Challenges III




- What if
  - Someone tries to break into your system to steal data?
  - … sniffs for information?
  - … your customer orders something and doesn’t accept the delivery saying he didn’t?
  - **Security**
- Or
  - You are so successful that millions of people are visiting your online store at the same time?
  - **Scalability**

# Business Example – Challenges IV

- When building the system…
  - Do you want to write the whole software on your own (network, database,…)?
  - What about updates, new technologies?
  - **Reuse** and **Openness** (Standards)

# Overview Challenges I

- Heterogeneity
  - Heterogeneous components must be able to interoperate
- Distribution transparency
  - Distribution should be hidden from the user as much as possible
- Fault tolerance
  - Failure of a component (partial failure) should not result in failure of the whole system
- Scalability
  - System should work efficiently with an increasing number of users
  - System performance should increase with inclusion of additional resources

# Overview Challenges II

- Concurrency
  - Shared access to resources must be possible
- Openness
  - Interfaces should be publicly available to ease inclusion of new components
- Security
  - The system should only be used in the way intended

# Heterogeneity

- Heterogeneous components must be able to interoperate across different:
  - Operating systems
  - Hardware architectures
  - Communication architectures
  - Programming languages
  - Software interfaces
  - Security measures
  - Information representation


# Distribution Transparency I

- ISO Reference Model for Open Distributed Processing (ODP) identifies the following forms of transparencies:
- Access transparency
  - Access to local or remote resources is identical
  - E.g. Network File System / **Dropbox**
- Location transparency
  - Access without knowledge of location
  - E.g. separation of domain name from machine address.
- Failure transparency
  - Tasks can be completed despite failures
  - E.g. message retransmission, failure of a Web server node should not bring down the website

# Distribution Transparency II

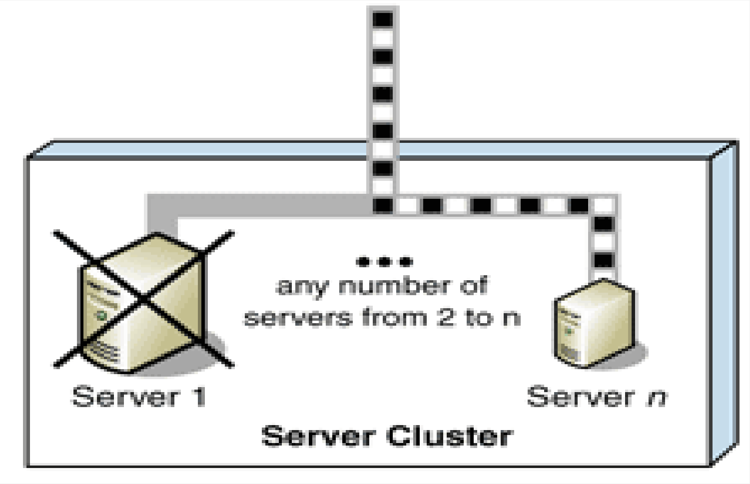
- Replication transparency
  - Access to replicated resources as if there was just one. And provide enhanced reliability and performance without knowledge of the replicas by users or application programmers.
- Migration (mobility/relocation) transparency
  - Allow the movement of resources and clients within a system without affecting the operation of users or applications.
  - E.g. switching from one name server to another at runtime; migration of an agent/process from one node to another.

# Distribution Transparency III

- Concurrency transparency
  - A process should not notice that there are other sharing the same resources
- Performance transparency:
  - Allows the system to be reconfigured to improve performance as loads vary
  - E.g., dynamic addition/deletion of components, switching from linear structures to hierarchical structures when the number of users increase
- Scaling transparency:
  - Allows the system and applications to expand in scale without changes in the system structure or the application algorithms.
- Application level transparencies:
  - Persistence transparency
    - Masks the deactivation and reactivation of an object
  - Transaction transparency
    - Hides the coordination required to satisfy the transactional properties of operations

# Fault Tolerance

- Failure: an offered service no longer complies with its specification (e.g., no longer available or very slow to be usable)
- Fault: cause of a failure (e.g. crash of a component)
- Fault tolerance: no failure despite faults i.e., programmed to handle failures and hides them from users.



# Fault Tolerance Mechanisms

- Fault detection
  - Checksums, heartbeat, …
- Fault masking
  - Retransmission of corrupted messages, redundancy, …
- Fault toleration
  - Exception handling, timeouts,…
- Fault recovery
  - Rollback mechanisms,…

# Scalability I

- System should work efficiently at many different scales, ranging from a small Intranet to the Internet
- Remains effective when there is a significant increase in the number of resources and the number of users

# Scalability II

- Challenges of designing scalable distributed systems:
  - Cost of physical resources
    - Cost should linearly increase with system size
  - Performance Loss
    - For example, in hierarchically structure data, search performance loss due to data growth should not be beyond O(*log n*), where n is the size of data
  - Preventing software resources running out:
    - Numbers used to represent Internet addresses (32 bit->64bit)
    - Y2K-like problems
  - Avoiding performance bottlenecks:
    - Use of decentralized algorithms (centralized DNS to decentralized)

# Concurrency

- Provide and manage concurrent access to shared resources:
  - Fair scheduling
  - Preserve dependencies (e.g. distributed transactions -- buy a book using Credit card, make sure user has sufficient funds prior to finalizing order )
  - Avoid deadlocks

# Openness and Interoperability

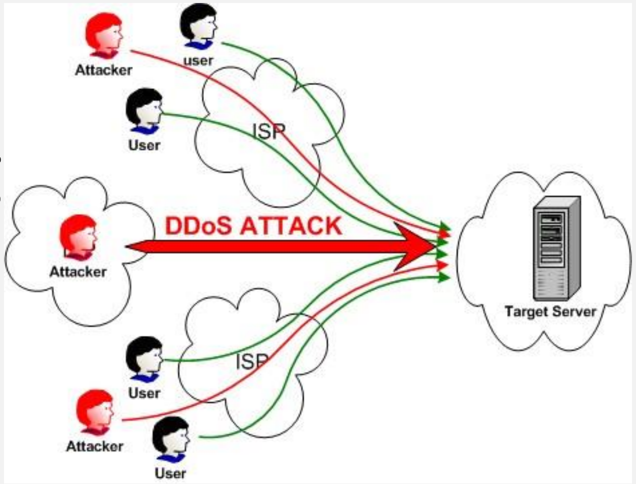
- An open system –key interfaces need to be published!
- An open distributed system has:
  - uniform communication mechanism
  - published interfaces to shared resources
- Open DS - heterogeneous hardware and software, possibly from different vendors, but conformance of each component to published standard must be tested and verified for the system to work correctly
- Open spec/standard developers - communities:
  - ANSI, IETF, W3C, ISO, IEEE, OMG, Trade associations,...

# Security I

- Resources are accessible to authorized users and used in the way they are intended
- Confidentiality
  - Protection against disclosure to unauthorized individual information
  - E.g. ACLs (access control lists) to provide authorized access to information
- Integrity
  - Protection against alteration or corruption
  - E.g. changing the account number or amount value in a money order

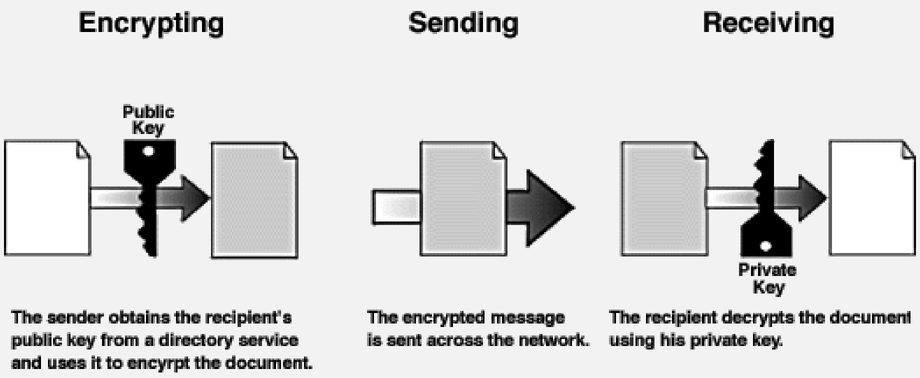
# Security II

- Availability
  - Protection against interference targeting access to the resources.
  - E.g. denial of service (DoS, DDoS) attacks
- Non-repudiation
  - Proof of sending / receiving an information
  - E.g. digital signature



# Security Mechanisms

- Encryption
  - E.g. Blowfish, RSA




- Authentication
  - E.g. password, public key authentication
- Authorization
  - E.g. access control lists

# Summary of DS

- Distributed Systems are everywhere
- Internet enables users throughout the world to access its (application) services from anywhere
- Resource sharing is the main motivating factor for constructing distributed systems
- Construction of DS produces many challenges:
  - Heterogeneity, Openness, Security, Scalability, Failure handling, Concurrency, and Transparency
- Distributed systems enable globalization:
  - Community (Virtual teams, organizations, social networks)
  - Science (e-Science)
  - Business (..e-Banking..)
  - Entertainment (YouTube, e-Friends)

# References

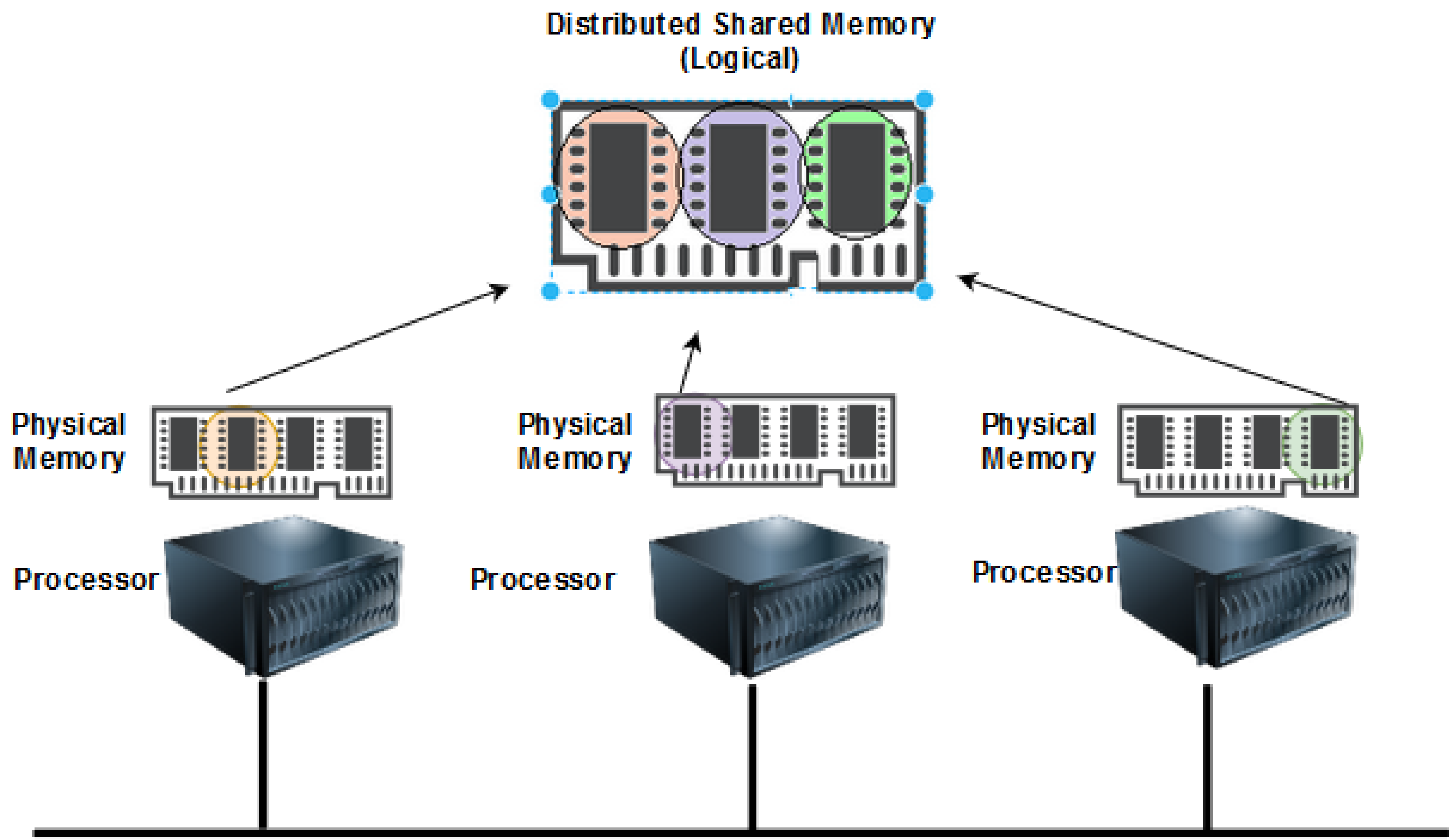

- https://mtjagtap.files.wordpress.com/2018/08/slides10-2.pdf
- http://www.cloudbus.org/652/LectureSlides.html
- Coulouris, G. (2012). Distributed systems. Boston: Addison-Wesley

# Distributed Shared Memory (DSM) Approaches

# How to share data between distributed processor nodes?

- Distributed Shared Memory
  - It's the abstraction to share data between distributed computers
- It enables access across multiple distributed physical memory to be accessed as single shared memory

# DSM Abstraction



# Working of DSM

- DSM enables easy access of individual computer shared data items
- DSM run time support sends updates as messages between computers
- Each computer has a local copy of recently accessed data items stored in DSM

# …Continued…

- Shared memory request for a non-local piece of data is raised
- Single copy of data fetched and given to the requested system
- If multiple machines access the data at the same time, synchronization primitive like semaphore is used to handle the situation

# …Continued

- Read(shared variable)
- Write (data, shared variable)

# Issues related to DSM Semantics

- Structure and granularity – data shared at the bit, word or page level
- Consistency – If multiple requests for a single data and each machine tries to update it, consistency should be maintained. This involves cache coherence like solution
- Heterogeneity – Accommodating different data representations of different machines, languages and OS
- Scalability – Bus latency, Increased broadcast messages

# DSM Versus Message Passing

- No marshaling of messages in DSM where as messages are marshalled and unmarshalled in message passing
- Synchronization in DSM is by constructs like semaphore/locks where as in message passing it is by message passing primitives
- DSM→ processes can communicate with non-overlapping lifetimes where as in message passing processes communicate at the same time
- Efficiency in DSM heavily depends on the pattern of Data access by multiple machines where as the same is not true for message passing

# DSM features

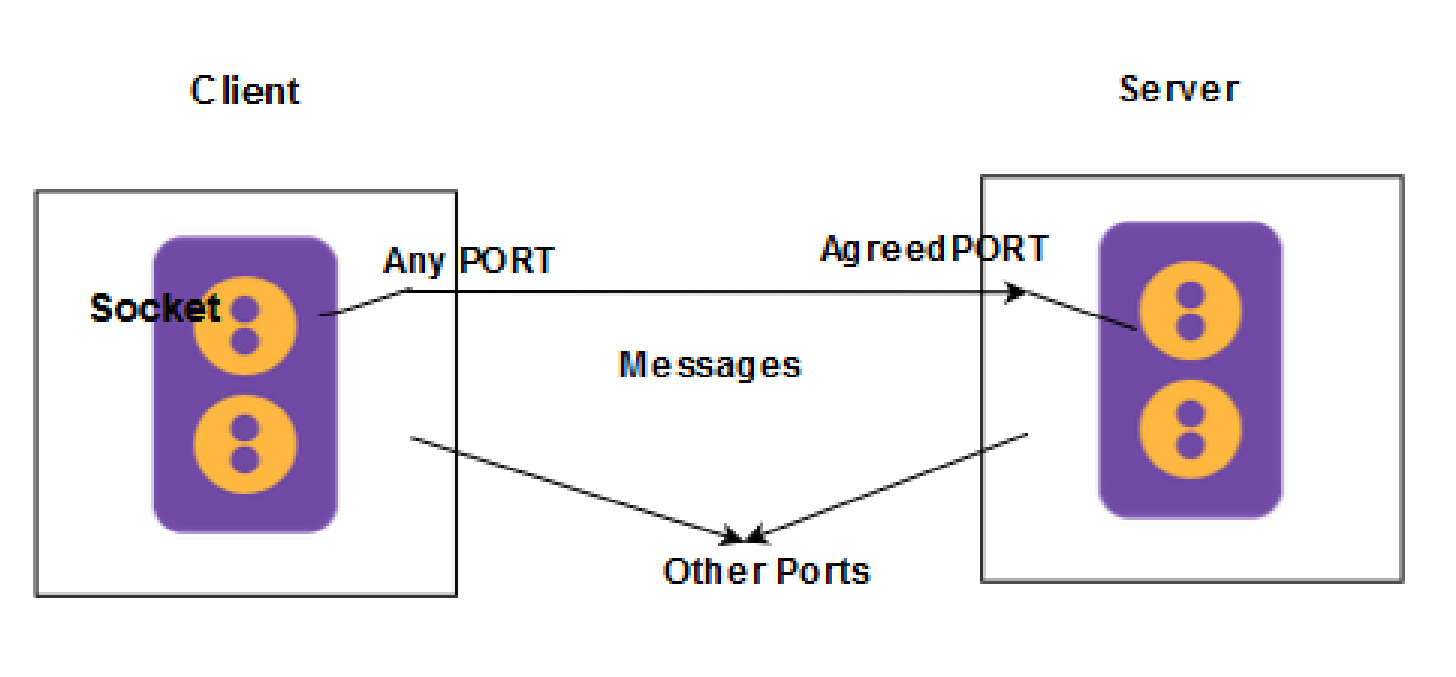
- Space- uncoupled
- Time – uncoupled
- State based service
- Used for parallel and distributed computation
- Limited scalability
- Not associative

# Message Passing
# &
# Programming Using the Message-Passing Paradigm

# Message Passing Interface (MPI)

- Standard API for message passing operations
- Includes synchronous and Asynchronous variants
- To communicate, one process sends a message (a sequence of bytes) to a destination and another process at the destination receives the message.

# Overview


Client
Server
Any PORT
Agreed PORT
Socket
Messages
Other Ports

# Synchronous and Asynchronous Message Passing

- Queue is associated with each message destination
- Sending process adds message to remote queue
- Receiving process reads from local queue
- Synchronous → Send and receive are blocking
- Asynchronous → Send is non-blocking ; Receive can be blocking or non-blocking

# External Data Representation

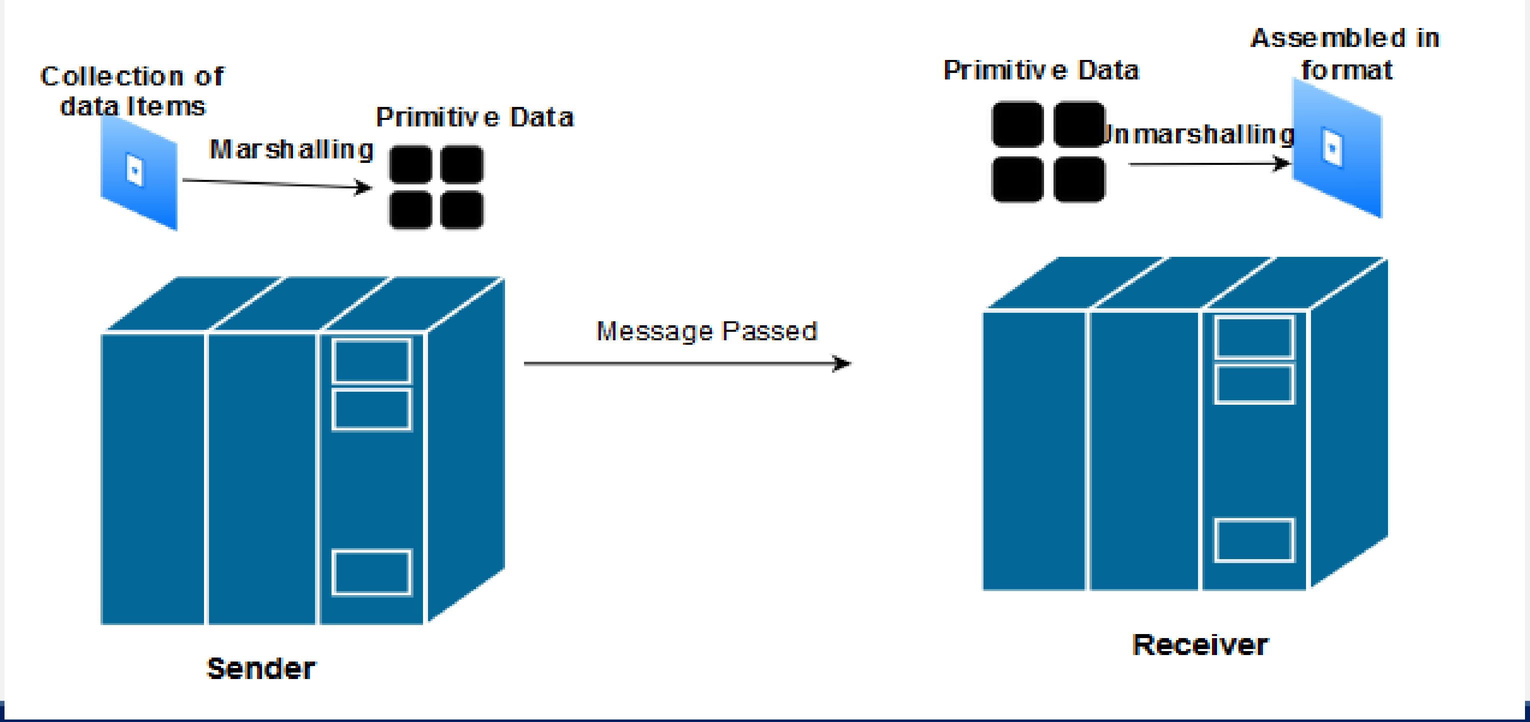
- Information stored in processes as data structures
- Data structures can be multi dimensional but has to be flattened before communication
- Similarly upon receiving has to be rebuilt

# Two ways to agree upon the format

- Both the machines will follow the same format
- Values are transmitted in sender’s format.
- Along with the values, data format is also sent to the reciever

# Marshalling

- Process of taking collection of data items and assembling them in order to transmit
- Unmarshalling is the reverse of marshalling



# Simple MPI Program

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

int main(int argc, char** argv) {
   // Initialize the MPI environment
   MPI_Init(NULL, NULL);

   // Get the number of processes
   int world_size;
   MPI_Comm_size(MPI_COMM_WORLD, &world_size);

   // Get the rank of the process
   int world_rank;
   MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

# ...Continued

```
// Get the name of the processor
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Get_processor_name(processor_name, &name_len);

    // Print off a hello world message
    printf("Hello world from processor %s, rank %d out of %d
processors\n",
           processor_name, world_rank, world_size);

    // Finalize the MPI environment.
    MPI_Finalize();
}
```

# Types of Communication in MPI

- Point to Point

- Group (or) Collective

# Point to Point Communication

- MPI_Send()
- MPI_Recv()
- MPI_Isend()
- MPI_Irecv()
- MPI_Probe()
- MPI_Wait()

# Broadcasting with MPI Send and Receive Sample

```
void my_bcast(void* data, int count, MPI_Datatype datatype, int root,  MPI_Comm communicator) {
  int world_rank;
  MPI_Comm_rank(communicator, &world_rank);
  int world_size;
  MPI_Comm_size(communicator, &world_size);

  if (world_rank == root) {
    // If we are the root process, send our data to everyone
    int i;
    for (i = 0; i < world_size; i++) {
      if (i != world_rank) {
        MPI_Send(data, count, datatype, i, 0, communicator);
      }
    }
  } else {
    // If we are a receiver process, receive the data from the root
    MPI_Recv(data, count, datatype, root, 0, communicator,
             MPI_STATUS_IGNORE);
  }
}
```

# Compare MPI_Bcast and MPI send, receive (Complete the code)

```
for (i = 0; i < num_trials; i++) {
  // Time my_bcast
  // Synchronize before starting timing
  MPI_Barrier(MPI_COMM_WORLD);
  total_my_bcast_time -= MPI_Wtime();
  my_bcast(data, num_elements, MPI_INT, 0,
MPI_COMM_WORLD);
  // Synchronize again before obtaining final time
  MPI_Barrier(MPI_COMM_WORLD);
  total_my_bcast_time += MPI_Wtime();
```

```
// Time MPI_Bcast
 MPI_Barrier(MPI_COMM_WORLD);
 total_mpi_bcast_time -= MPI_Wtime();
 MPI_Bcast(data, num_elements,
MPI_INT, 0, MPI_COMM_WORLD);
 MPI_Barrier(MPI_COMM_WORLD);
 total_mpi_bcast_time += MPI_Wtime();
}
```

# MPI Non Blocked Point to Point Communication

- MPI_Isend and MPI_Irecv will not wait for the buffer data to be copied
- It will attach a pointer to the message transfer and immediately return
- Use MPI_Wait to ensure proper communication between send and receive

# Sample

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char *argv[])
{
   int myid, numprocs, left, right;
   int buffer[10], buffer2[10];
   MPI_Request request, request2;
   MPI_Status status;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,
&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,
&myid);
   right = (myid + 1) % numprocs;
   left = myid - 1;
if (left < 0)
      left = numprocs - 1;
   MPI_Irecv(buffer, 10, MPI_INT, left,
123, MPI_COMM_WORLD, &request);
   MPI_Isend(buffer2, 10, MPI_INT, right,
123, MPI_COMM_WORLD, &request2);
   MPI_Wait(&request, &status);
   MPI_Wait(&request2, &status);
   MPI_Finalize();
   return 0;
}
```

# MPI Collective Communications

- MPI_BCAST
- MPI_BARRIER
- MPI_SCATTER
- MPI_GATHER

# MPI_BCAST

```
main()
        {
                ....
                data =1;
                MPI_Bcast(data, num_elements,
                MPI_INT, 0, MPI_COMM_WORLD);
                //print the value of data inside a if
condition that                          checks the rank (other
than the root)


        }
```

# MPI_Scatter

- Designated root process sending data to all processes in a communicator
- MPI_Bcast → Same data
- MPI_Scatter → chunk of data from different processes

# Syntax

- MPI_Scatter( void* send_data, int send_count, MPI_Datatype send_datatype, void* recv_data, int recv_count, MPI_Datatype recv_datatype, int root, MPI_Comm communicator);

# Example: Root process generates random numbers and scatters it among multiple slaves

```
float create_rand_nums(int  n)
                {
                 float *rnd=(float *) malloc(sizeof(float)*n);
                for i= 0 to n
                                {
                    rnd[i]=rand();
                                 }
         return rnd;
                 }
```

# …Continued

```
main()
        {
        ….
        int id;
         int p;
        MPI_Comm_rank(MPI_COMM_WORLD, &id);
        MPI_Comm_size(MPI_COMM_WORLD, &p);
          float *rand_nums = NULL;
            if (id== 0) {
                    rand_nums = create_rand_nums(int  *p);
                   }
        float *sub_rand_nums = (float *)malloc(sizeof(float) * p);
            MPI_Scatter(rand_nums, num_elements_per_proc, MPI_FLOAT, sub_rand_nums, 1,
MPI_FLOAT, 0,           MPI_COMM_WORLD);
             …..
            }
```

# MPI_Gather

- MPI_Gather( void* send_data, int send_count, MPI_Datatype send_datatype, void* recv_data, int recv_count, MPI_Datatype recv_datatype, int root, MPI_Comm communicator);

# MPI_AllGather

- It is MPI_Gather followed by MPI_Bcast

# Case Study (RPC and Java RMI)

# RMI

- The **RMI** (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java.
- The RMI allows an object to invoke methods on an object running in another JVM.
- The RMI provides remote communication between the applications using two objects *stub* and *skeleton*.

# …Continued…

- A **remote object** is an object whose method can be invoked from another JVM.
- stub
  - The stub is an object, acts as a gateway for the client side.
  - All the outgoing requests are routed through it.
  - It resides at the client side and represents the remote object.

# What happens when the caller invokes method on the stub object?

- It initiates a connection with remote Virtual Machine (JVM),
- It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
- It waits for the result
- It reads (unmarshals) the return value or exception, and
- It finally, returns the value to the caller.

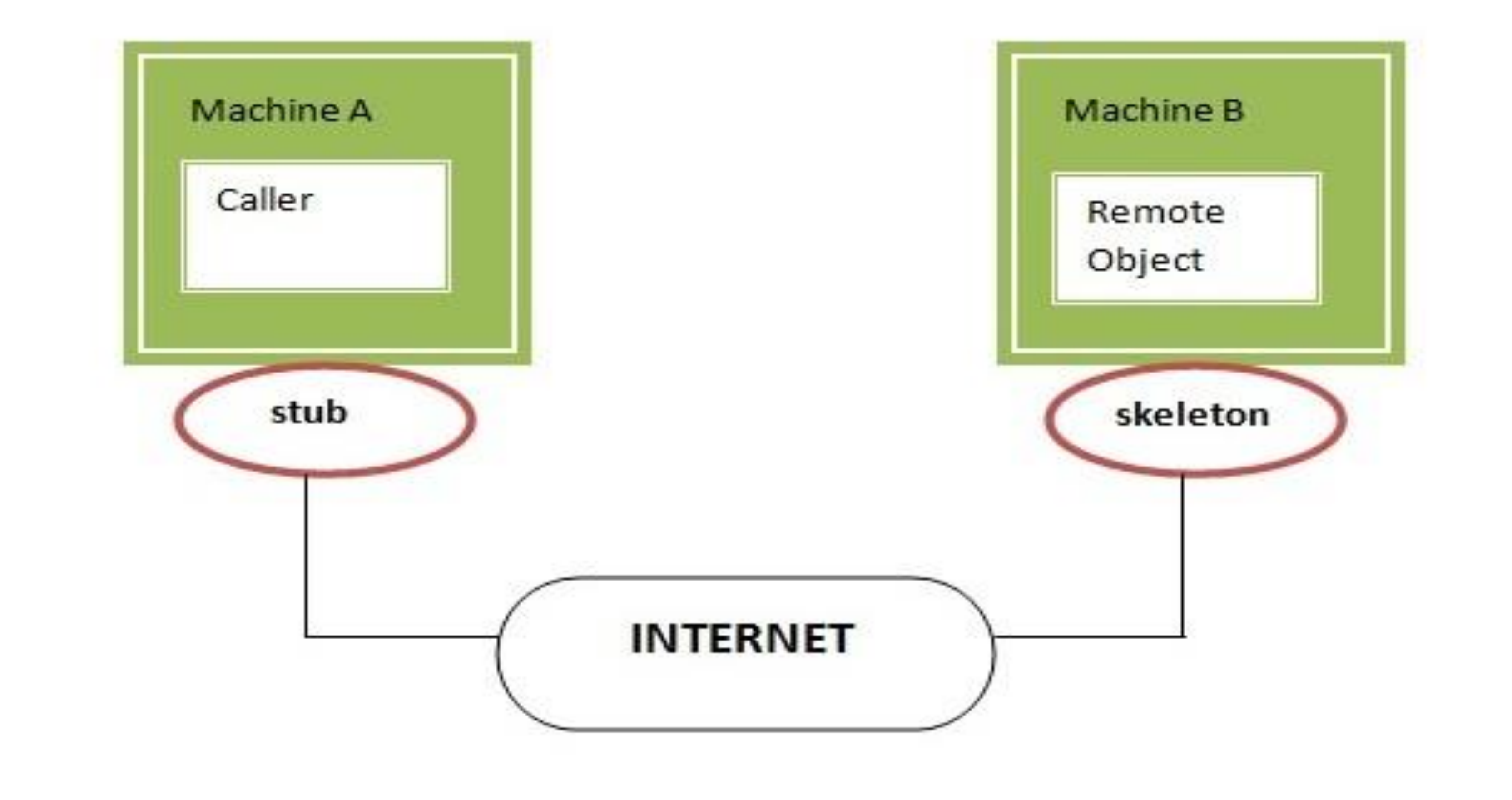
# skeleton

- The skeleton is an object, acts as a gateway for the server side object.
- All the incoming requests are routed through it.

# Upon receiving the incoming request

- It reads the parameter for the remote method
- It invokes the method on the actual remote object, and
- It writes and transmits (marshals) the result to the caller.

# …Continued…



Machine A
Caller
stub
Machine B
Remote
Object
skeleton
INTERNET

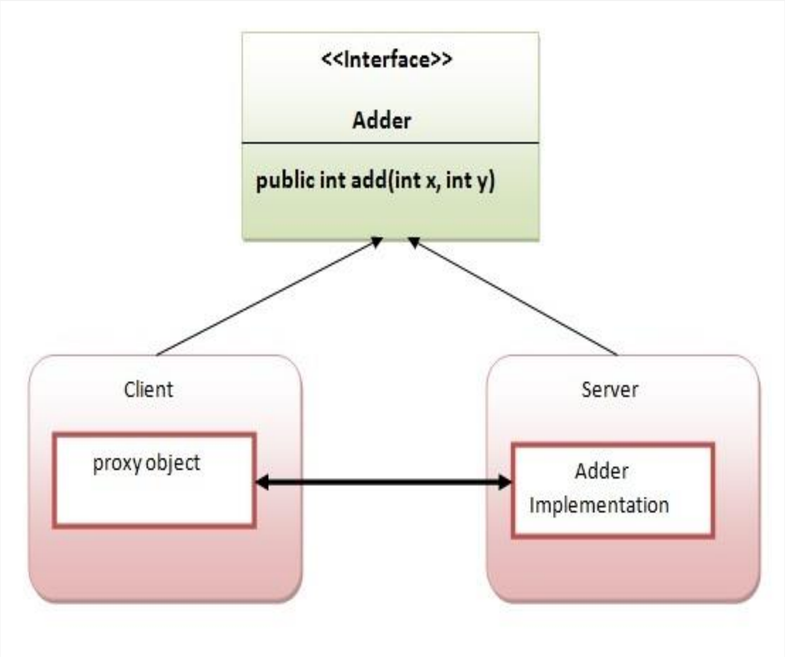
# Distributed applications Requirements

- If any application performs these tasks, it can be distributed application.
  - The application need to locate the remote method
  - It need to provide the communication with the remote objects, and
  - The application need to load the class definitions for the objects.
- The RMI application have all these features, so it is called the distributed application.

# Java RMI Example

- 6 steps to write the RMI program
  - Create the remote interface
  - Provide the implementation of the remote interface
  - Compile the implementation class and create the stub and skeleton objects using the rmic tool
  - Start the registry service by rmiregistry tool
  - Create and start the remote application
  - Create and start the client application

- The client application need only two files, remote interface and client application.
- In the rmi application, both client and server interacts with the remote interface.
- The client application invokes methods on the proxy object, RMI sends the request to the remote JVM.
- The return value is sent back to the proxy object and then to the client application.



# 1) create the remote interface

```
import java.rmi.*;
public interface Adder extends Remote
{
public int add(int x,int y)throws RemoteException;
}
```

# 2) Provide the implementation of the remote interface

```
import java.rmi.*;
import java.rmi.server.*;
public class AdderRemote extends UnicastRemot
eObject implements Adder{
AdderRemote()throws RemoteException{
super();
}
public int add(int x,int y){return x+y;}
}
```

# 3) create the stub and skeleton objects using the rmic tool.

- The rmic tool invokes the RMI compiler and creates stub and skeleton objects
- rmic AdderRemote

# 4) Start the registry service by the rmiregistry tool

- start the registry service by using the rmiregistry tool.

rmiregistry 5000

5000 – port number

# 5) Create and run the server application

```
import java.rmi.*;
import java.rmi.registry.*;
public class MyServer{
public static void main(String args[]){
try{
Adder stub=new AdderRemote();
Naming.rebind("rmi://localhost:5000/sonoo",stub);
}catch(Exception e){System.out.println(e);}
}
}
```

# 6) Create and run the client application

```
import java.rmi.*;
public class MyClient{
public static void main(String args[]){
try{
Adder stub=(Adder)Naming.lookup("rmi://localhost:5000/sonoo");
System.out.println(stub.add(34,4));
}catch(Exception e){}
}
}
```