CES-27 & CE-288
Distributed Programming

Chapter 1 – Introduction to distributed systems

Celso HIRATA, hirata@ita.br

JULIANA de Melo Bezerra, juliana@ita.br

Instituto Tecnológico de Aeronáutica

Agenda

1. Basic Concepts

1.1. What is **a distributed system**?

1.2. **Distribution**

1.3. What does a distributed system contain?

1.4. **Characteristics** of **distributed systems**

1.5. **Desirable** characteristics of **distributed algorithms**

1.6. Why do we build **distributed systems?**

2. Desirable Properties of distributed systems

2.1. Fault-tolerant

2.2. Highly available

2.3. Recoverable

2.4. Consistent

2.5. Scalable

2.6 .Predictable performance

2.7. Secure

3. Distributed system failures

3.1. **Categories** of failures: **hardware** and s**oftware**

3.2**. Residual bugs**

3.3**. Types** of **failures**

3.4**. Design** for **failure**

4. Design of distributed systems

5.1. Process limitations

5.2. Limiting the scope

5. The 8 Fallacies

5.1. The network is **reliable**.

5.2. **Latency** is zero.

5.3. **Bandwidth** is infinite.

5.4. The network is **secure**.

5.5. **Topology** does not change.

5.6. There is one **administrator**.

5.7. **Transport cost** is zero.

5.8. The network is **homogeneous**

Agenda

6. Distributed programs and languages	7. Logical topologies
6.1. Distributed **program** 6.2. Distributed programming **languages** 6.3. **CONIC** 6.5.1. Process **structure** 6.5.2. The **interface** block 6.5.3. The command **LOOP-SELECT** 6.5.4. Multiple **interfaces** 6.5.5. The **SELECT** command and object oriented languages 6.5.6. **Process** description 6.5.7. **Graphics** representation 6.5.8. **Components** in conic 6.5.9. **Processes** interactions	7.1. **Ring** 7.2. **Star** 7.3. **Tree** 7.4. **Completed** connected

1. Basic Concepts

What is a distributed system?

Program

Distr. Program

Process

Message

Tread

Operating System

Package

Protocol

Component

Host

Network

Distributed System

What is a distributed system?

- **Program** is the code you write.
- **Distributed program** is a set of processes that exchange messages.
- **Process** is what you get when you run it, typically managed by an OS.
- **Thread** of execution is a sequence of programmed instructions, typically managed within a process.

What is a distributed system?

- **Message** is used to communicate between processes.
- **Packet** is a fragment of a message that might travel on a wire or air.
- **Protocol** is a **formal description** of **message formats** and the **rules** that **processes must follow** in order to exchange messages.
- **Network** is the infrastructure that links **computers**, **workstations**, **terminals**, **servers**, etc.
  - **Routers, switches, links, etc.**

What is a distributed system?

- A **component** can be a **process** or any piece of **hardware** required to:
  - ✓ **run** a process,
  - ✓ support **communications** between processes,
  - ✓ **store** data,
- ✓ Abstraction in a architecture to ease development.

What is a distributed system?

- A **distributed system** is is an **application** that executes a **collection of protocols** to coordinate the actions of multiple **processes** on a **network**
- All application **components cooperate together** to perform a single or small set of **related tasks**.

Hardware distribution

- The **system** is:
  - ✓ **Distributed** with **autonomous** systems
  - ✓ No **master-slave** control, yet **cooperative**.
  - ✓ **Serial communication** among systems
    - **Failures** and **delays** are **possible**

Software Distribution

- The **software** is:
  - ✓ **Distributed** and **parallel**
  - ✓ **Decentralized control** with **resources sharing**.
    - **cooperative** and/or **competitive**
  - ✓ **Communication** and **synchronization** by the use of **messages**.

What does a distributed system contain?

- **Autonomous processors** and/or **data stores** that support **processes** and/or **databases**
  - They interact in order to **cooperate** to achieve a **common goal**.

Characteristics of distributed systems

Communication by **message passing** in a **shared** communication **structure**

- ✓ There is **no shared memory**
- ✓ Inter-process **message communication** susceptible to **delay** and **failure**
- ✓ **Time not null** between an **event** and its acknowledgement
- ✓ There might no be **ordering of messages**

Characteristics of distributed systems

Some control at **system level**

- Inter-process **cooperation** and **management**
- **System state** is **partitioned** and **distributed**
- Control of the system must use **partial information**
  - ✓ There is **no** coherent **global view**!

Desirable characteristics of distributed algorithms

1. **Partitioning** degree:
    1. Asymmetry: different code
    2. **Textual** symmetry: same code, different process-id, different roles
2. **Fault** resilience
3. **Minimum** of **connection** properties
4. **Performance**
5. **Global** and **local states**

Desirable characteristics of distributed algorithms

- C**omplete symmetry:** same code, same (no) process-id, same role.
  - S**ome degree** of **asymmetry** is usually required in order to solve conflicts: different process-id
  - In that way, textual **symmetry** is the most **common**.

Desirable characteristics of distributed algorithms

2. **Fault resilience**

✓ We would like our **algorithms** were resilient to any **process failure**

- Hence the desire for **symmetry**

3. **Connection properties**

✓ We would like to have **algorithms** that request a **minimum** of **connection properties**.

- This results in **resilient algorithms** that can use simple **networks**

Desirable characteristics of distributed algorithms

4. **Performance**
   - ✓ We would like that **algorithms minimize** the **traffic** of **messages** and reduce the **response time** of the **algorithm**.
5. **Global and local states**
   - ✓ We would like that individual **processes** were able to **make decisions** based on **local knowledge** without the need of **global knowledge**.
   - ✓ This reduces the **traffic of message** and improve **resilience**.

Why do we build distributed systems?

- One advantage is the ability to **connect remote users** with **remote resources** in an **open** and **scalable** way
  - ✓ By **open**
    - Each **component** is **continually open** to **interaction** with other **components**.
  - ✓ By **scalable**
    - The **system** can easily be altered to accommodate **changes** in the **number** of **users**, **resources** and computing **entities**.

Architectures

- **Peer-to-peer:** all responsibilities (provide the service or manage the network resources) are uniformly divided among all peers. Peers can be seen as both clients and servers.
- **Client–server**: architectures where smart clients contact the server for data then format and display it to the users.
- **Three-tier:** architectures that move the client intelligence to a middle tier so that stateless clients can be used.
- **n-tier**: architectures that refer typically to web applications which further forward their requests to other enterprise services.

Cloud computing

- **Cloud computing** is the on-demand availability of computer system resources, especially data storage and computing power, without direct active management by the user.
- Used to describe data centers available to many users over the Internet.

Fog computing

- **Fog computing** as a horizontal, physical or virtual resource paradigm that resides between smart end-devices and traditional **cloud computing or data center.**
- Explores the proximity to achieve better quality of service (QoS) (performance, reliability, service)
- Supports the **Internet of Things (IoT)** (phones, wearable health monitoring devices, connected vehicle, augmented reality devices such as the Google Glass).

Edge Computing

- Distributed computing paradigm which brings computation and data storage closer to the location where it is needed, to improve response times and save bandwidth.
- Refers to extending cloud computing **to the edge of an enterprise's network**.
- Edge computing is typically referred to the location where services are instantiated,

Other systems

- Cooperative and collaborative systems
- Cyber physical systems: UTM (UAS), ATM (Air), Autonomous Vehicle, Healthcare systems, Smart city systems

2. Desirable properties of distributed systems

Desirable properties of distributed systems

- In order to be **useful**, a **distributed system** must be **exhibit desirable properties (non-functional requirements)**
  - ✓ difficult goal to achieve because of the **complexity** of the **interactions** between **simultaneously running components**.

Desirable properties of distributed system

- Each **characteristic** discussed in the next **slides**.

    1. **Fault-tolerant**
    2. Highly **available**
    3. **Recoverable**
    4. **Consistent**
    5. **Scalable**
    6. Predictable **performance**
    7. **Secure**
    8. **Safety and others**

Desirable properties of distributed system

1. **Fault-Tolerant**
   - ✓ It can **recover** from **component failures** without performing incorrect actions. (Braking system)
2. **Highly Available**
   - ✓ It can **restore operations**, permitting it to **resume** providing **services** even when some components have **failed**. (Online banking service)

Desirable properties of distributed system

3. **Recoverable**
   - ✓ Failed **components** can **restart themselves** and **rejoin the system**, after the cause of **failure** has been repaired. (e.g, Satellite)
4. **Consistent**
   - ✓ The system can **coordinate actions** by multiple **components** often in the presence of **concurrency** and **failure**. (Banking transfer)

Desirable properties of distributed system

5. Scalable

- ✓ It **continues to operate correctly** even if some aspect of the system is scaled to a **larger size**.
- ✓ **For example**, we might **increase** the **number of users** or **servers**, or **overall load** on the system. (Video streaming)

Desirable properties of distributed system

6. **Predictable Performance**
   - ✓ The ability to provide desired **responsiveness** in a **timely manner**.
7. **Secure**
   - ✓ The system **authenticates access** to **data** and **services and provides confidentiality and integrity**.
   - ✓ Data privacy
8. **Safety** (Avoid accidents - safety-critical systems such as autonomous vehicle)

3. Distributed systems failures

Categories of failures

- Handling **failures** is an **important theme** in **distributed systems design**

- Two **categories**:
  - ✓ **hardware** and **software.**

Hardware failures

- **Hardware failures** were a dominant concern until the late 80's,
  - but since then internal **hardware reliability** has improved enormously

Software failures

- **Software failures** are a significant issue in **distributed systems**.
  - ✓ Software **bugs** account for a substantial fraction of **unplanned downtime** (estimated at **25-35%**)
  - ✓ Even with **rigorous testing**

Residual bugs

- **Residual bugs** in **mature systems** can be classified into **two main categories**.
  - ✓ **Bohr bug**: **bug** that **does not disappear or alter** its **characteristics** when it is **researched**. It is easier to find.
  - ✓ **Heisenberg bug**: **bug** that seems to **disappear or alter** its **characteristics** when it is **observed** or **researched**
    - **M**ore common in distributed systems than in local systems.
    - Difficult to o**btain** a coherent and **comprehensive view** of the **interactions** of **concurrent processes**

Types of failures

- There are **seven types** of **failures** in **distributed systems**

1. **Halting** failures
2. **Fail-stop**
3. **Omission** failures
4. **Network** failures
5. **Network partitioning** failures
6. **Timing** failures
7. **Byzantine** failures

Types of failures

1. **Halting failures**

- ✓ A **component** simply **stops**.
- ✓ There is **no way to detect the failure** except by **timeout**
  - It either stops sending **"I'm alive" (heartbeat) messages** or fails to **respond to requests**.
- ✓ Your **computer freezing** is a **halting failure**.

Types of failures

2. Fail-stop

- ✓ A **halting failure** with some kind of **notification** to other **components**.
  - A **network file server telling** its clients it is about to **go down** is a **fail-stop**.

Types of failures

3. **Omission failures**
   - ✓ **Failure to send/receive messages** primarily **due to** lack of **buffering space**
     - which causes a **message** to be **discarded** with **no notification** to either the **sender** or **receiver**.
     - **E.g.** routers become **overloaded**.

4. **Network failures**:
   - ✓ A **network link** breaks.

Types of failures

5. **Network partition failure**:
   - ✓ A **network fragments into** two or more **disjoint sub-networks** within which messages can be sent, but between which **messages** are **lost**.
   - ✓ E.g. **network failure**.

Types of failures

6. **Timing failures**:
   - ✓ A **temporal property** of the system is **violated**.
   - ✓ For example, **clocks** on **different computers** which are used to coordinate processes are **not synchronized**;
   - ✓ Occur when a **message** is **delayed longer** than a threshold **period**, etc.

Types of failures

7. **Byzantine failures**:
   - ✓ This **captures** several types of **faulty behaviors** including **data corruption** or **loss**, **failures caused** by **malicious programs**, etc. (**There is intent**!)

Design for failure

- Our **goal** is to **design a distributed system** with **some of** the **characteristics** aforementioned.

- which means **we must design** for **failure**.
  - We **must** be careful to **make assumptions** about the **properties** of the **components** of a system.
  - The system can be very hard and/or expensive to build if we want to meet **all the desirable properties**.

4. Design of distributed systems

Design of distributed systems

- **Building** a **reliabl**e system that **runs over** an **unreliable communications network** seems like an impossible goal.

- **Distributed systems design** is, indeed, a challenging **endeavor**.

Process limitations

- A **process** knows its own **state**, and it knows what **state** other **processes** were in recently
  - however, the **processes** have **no** way of **knowing each other's current state**.
  - They **lack** the equivalent of **shared memory**.

- **Processes** also **lack** accurate **ways** to:
  - Detect **failure**
  - **Distinguish** a **local software/hardware failure** from a **communication failure**

Limiting the scope

- Generally we start considering a particular **type** of **distributed systems design, including an architecture.**
  - For instance:
    - **client-server model** with mostly **standard protocols**.
  - These **standard protocols** provide help with the **low-level details** of **reliable network communications**
    - makes our job easier ;)

5. The 8 fallacies

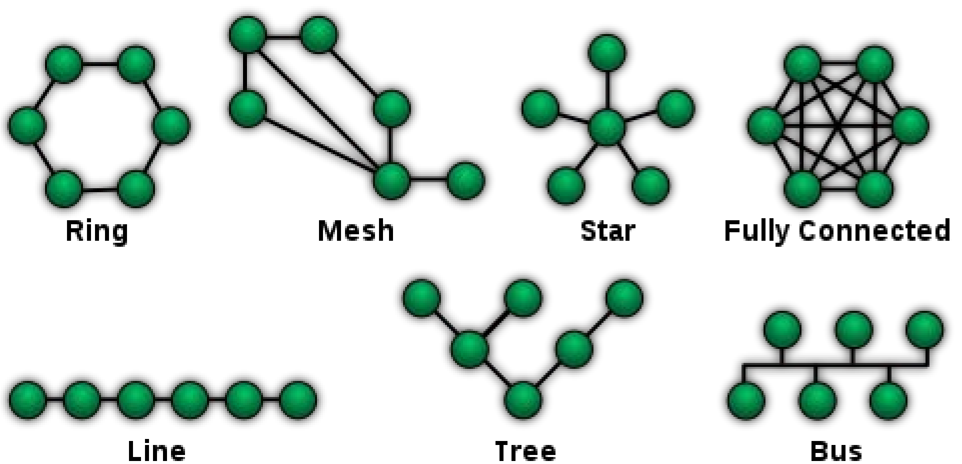
Fallacy - a mistaken belief, especially one based on unsound argument.

The 8 fallacies

1. The **network** is **reliable**
2. **Latency** is zero
3. **Bandwidth** is infinite
4. The network is **secure**
5. Topology does not change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

The 8 fallacies

5. **Topology** does not change
   - ✓ The different **configurations** that can be adopted in building **networks**
   - ✓ Topologies include **ring**, **bus**, **star, completed connected** or **meshed**.



The 8 fallacies

6. **There is one administrator**
   - ✓ Usually we have **several administrators** involved in a **distributed system**.
   - \- **Administrators** set **disk quotas** and limit **privileges**, **ports,** and **protocols**.
   - \- You need to help them **manage** your **applications**.

The 8 fallacies

7. **Transport cost** is zero
   - ✓ Going from the **application level** to **the transport level** is **not free**.
     - We have to do **marshaling** (serialize information into bits) to get data onto the wire which takes both computer **resources** and increases the **latency**
   - ✓ Costs for setting and running the network are not free
     - Costs for buying the **routers**, **securing** the network
     - Costs for **leasing the bandwidth** for internet connections
     - Costs for operating and maintaining the network running

The 8 fallacies

8. **Homogeneous network**
   - ✓ **Usually networks** run multiple **network protocols and** are **not homogeneous**
     - **One advice: do not rely on proprietary protocols and use standard technologies instead.**

6. Languages for Distributed Programs

Distributed programming languages

- **Distributed program = processes + messages**

- **Some programming languages**
  - **Conic/Rex, Darwin/Regis: Imperial College**
  - **CSP/Occam (Oxford): Transputers**
  - **CORBA**
  - **Java**
  - **C#**
  - **Go**

Distributed programming languages: requirements

- **Transparent** distribution

- **Type verification**

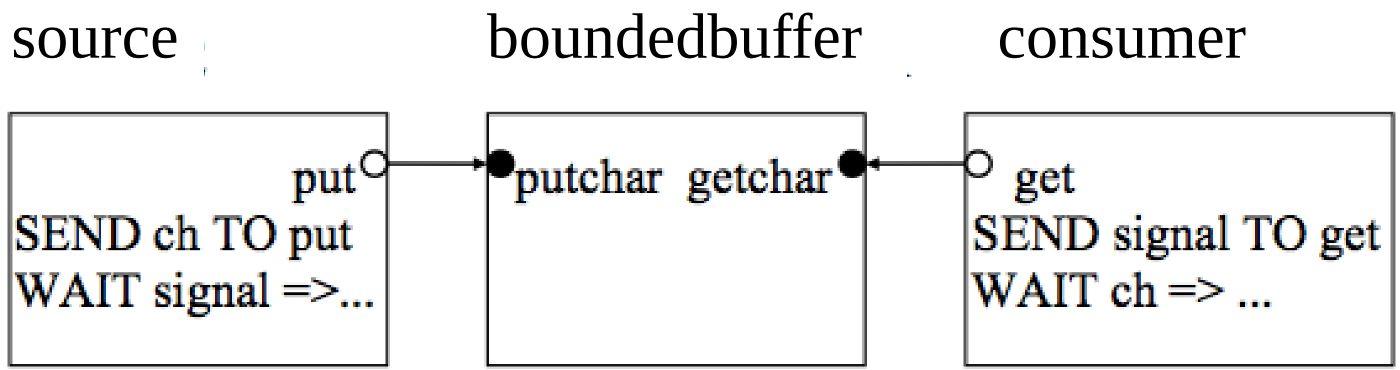
- **Configuration** support

Conic

- **Distributed programming language**
- **Conic** will be used to **describe processes** in this **course**
  - ✓ **Easy!** ☺
- In Conic, a distributed program is seen as two compiled parts:
  - ✓ **Module** that contains the **configuration** of tasks
  - ✓ **Tasks** (processes)

Conic: Graphical representation

- Graphical representation of a distributed program using a Conic Example
- Configuration of source, boundedbuffer, and consumer



○ referência de porta (cf referência de objeto)
● porta (cf objeto)

Components in Conic

```
component simple_distr_prog() {
    // provide port-interface<port type-of-message>;
    // require port-ref-interface<port type-of-message>;
    inst                          // instantiate the 3 components
        S: source;          // S is of type source
        B: boundedbuffer;
        C: consumer;
    Bind
        S.put – B.putchar;   // Connects put of S to putchar of B
        C.get – B.getchar;   // Connects get of C to getchar of B

}
```

Conic

- **Processes** (tasks)
  - ✓ **Sequential**
  - ✓ **Non-deterministic** reply to events
  - ✓ Local **data**
  - ✓ Communication transactions

Task structure: 3 blocks

TASK MODULE boundedbuffer;

interface

    // block with definitions of communication interfaces

    // where the messages are sent or received

data

    // variable definitions block

BEGIN

    // commands block

END

The interface block

- Examples of definition for the **interface** block:

  ENTRYPORT putchar: char **REPLY** signaltype;

  ENTRYPORT getchar: signaltype **REPLY** char;

- The first example defines an interface **putchar** that receives **message** of **char** type, and returns another **message** of the **pre-defined** type, **signaltype.**

The command block: LOOP-SELECT

- Inside a **command block**, usually the **LOOP-SELECT** command is used.
  - ✓ It allows that the **process** receives **messages** from **more than one** interface **concurrently** (multiple entry points).
  - ✓ boundedbuffer has two ports to receive messages: **putchar** and **getchar**

The command LOOP-SELECT

```
LOOP
  SELECT
    WHEN condition-to-receive-message-of-type-1
    RECEIVE variable-to-receive-message-of-type-1 FROM
            interface-1 REPLY reply-msg-1 =>
                  // commands
  OR
    WHEN condition-to-receive-message-of-type-2
    RECEIVE variable-to-receive-message-of-type-2 FROM
            interface-2 REPLY reply-msg-2 =>
                  // commands
  END;
END;
```

Note: signal is a pre-defined type in Conic

Multiple interfaces

- A **distributed process** that can receive messages from more than on **port** (**interface**) usually has the **following structure**:

LOOP

SELECT

RECEIVE$_1$ => PROC$_1$

OR RECEIVE$_2$ => PROC$_2$

…

OR RECEIVE$_n$ => PROC$_n$

END // Select

END // Loop

Task description in Conic

```
TASK MODULE boundedbuffer;
interface
        ENTRYPORT putchar: char REPLY signaltype;
        ENTRYPORT getchar: signaltype REPLY char;
data
   CONST poolsize = 100;
   VAR pool: ARRAY[1..poolsize] OF char;
   inp, outp: 1..poolsize;
   count     : 0..poolsize;
BEGIN
   inp := 1; outp := 1; count := 0;
   LOOP
      SELECT
         WHEN count < poolsize
         RECEIVE pool[inp] FROM putchar REPLY signal =>
                   inp := (inp MOD poolsize) + 1;
                   count := count + 1;
      OR
         WHEN count > 0
         RECEIVE signal FROM getchar REPLY pool[outp] =>
                   outp := (outp MOD poolsize) + 1;
                   count := count -1
      END;
   END;
END;
```

Running simple_distr_prog

- source sends 2 characters “I”, “T” to boundedbuffer
- consumer requests 3 chars from boundedbuffer
- source sends character “A”
- Initially count = 0, inp =1, outp=1, pool=“??????????…”
- The content of pool is given by the pointer outp and count
  - ✓ [Completear]

Running simple_distr_prog

- ✓ Initially count = 0, inp =1, outp=1, pool="??????????…"
- ✓ Receiving "I": inp=2, count=1, pool= "I???..."
- ✓ Receiving "T": inp=3, count=2, pool="IT???..."
- ✓ Receiving request: outp=2, count=1, pool="IT???..."
- ✓ Receiving request: outp=3, **count=0**, pool="IT???..."
- ✓ boundedbuffer does not receive request (**count=0**)
- ✓ Receiving "A": inp=4, count=1, pool="ITA??..."

  // enables to receive request
- ✓ Receiving request: outp=4, count=0, pool="ITA??..."

The SELECT command and object oriented languages

- Nowadays, the **some programming languages do not have** the **SELECT command**.
  - ✓ It can be implemented using **one thread of execution** for **each RECEIVE**.
    - Java, C#

The SELECT command and object oriented languages

- Define $\boldsymbol{t_1}$, $\boldsymbol{t_2}$, $\boldsymbol{t_n}$ as **threads** that **receive messages** on the **interfaces**.
- Each ***thread*** $\boldsymbol{t_i}$ is a **loop** that receives a message and process it.

Thread t_i()

LOOP

RECEIVE$_{\mathbf{i}}$;

PROC$_{\mathbf{i}}$;

END

The SELECT command and object oriented languages

- **SELECT** is implemented as **thread calls** to $\boldsymbol{t_1, t_2, ...,}$ $\boldsymbol{t_n}$ in **object oriented languages**

T_1 t_1 = new T_1();

T_2 t_2 = new T_2();

…

T_n t_n= new T_n();

t_1.start();

t_2.start();

...

t_n.start();

Processes interactions

- **Communication transactions**
  - **Distributed components** of a **distributed system** communicate in order to **cooperate** and **synchronize** their **actions**:
    - Exchange **information** in the form of **message** with type
    - **Synchronization**
      - One **message** without information **content** and with a goal of **synchronization** is called **signal**

Processes interactions

- **Unidirectional** primitives (asynchronous) at the **ports**:

  SEND (message) ----------> RECEIVE (message)

  The sender continues its processing after sending.

- **Bidirectional** at the **ports**:

  SEND (message) ----------->

  IN (message)

  SEND (reply)

  <-----------

  IN (reply)

Processes interactions

➢ **Bidirectional** at **input**:

CALL (message, reply) ---------->

ACCEPT (message)

REPLY (reply)

<-----------

cf. **REMOTE PROCEDURE CALL**(call params, return params)

Processes interactions

- **Unidirectional** (synchronous) at the **ports**:

OUT (message) ---------> **IN** (message)

The sender blocks its processing after sending until it receives a signal from receiver.

7. Logical topologies

Logical topologies

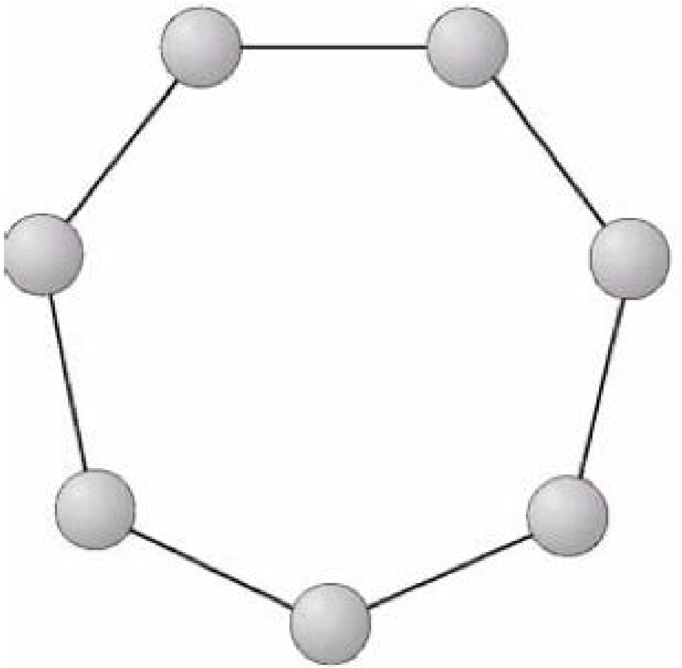
- **Connections** between **processes** usually assumed to be **one-to-one**.

- The **algorithms** use **one** the **topology** listed below :
  - ✓ **Ring**, **star**, **tree** or **completed connected**

- With some additions, one **logical topology** can **simulate any other**.

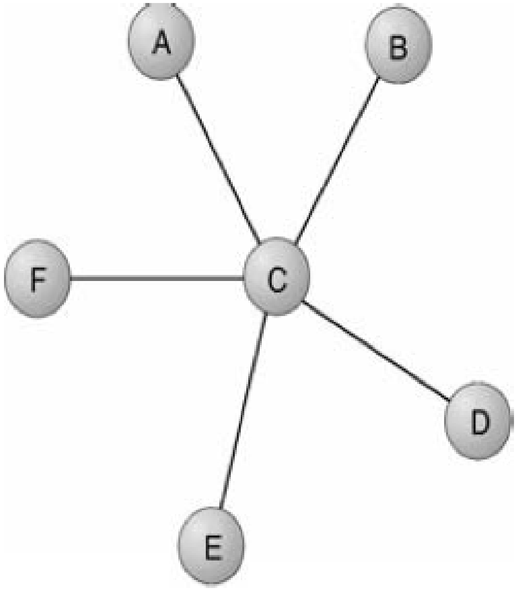
Ring

- One **process** only **communicates** with its **neighbors**
- **Uni** or **bi-directional.**



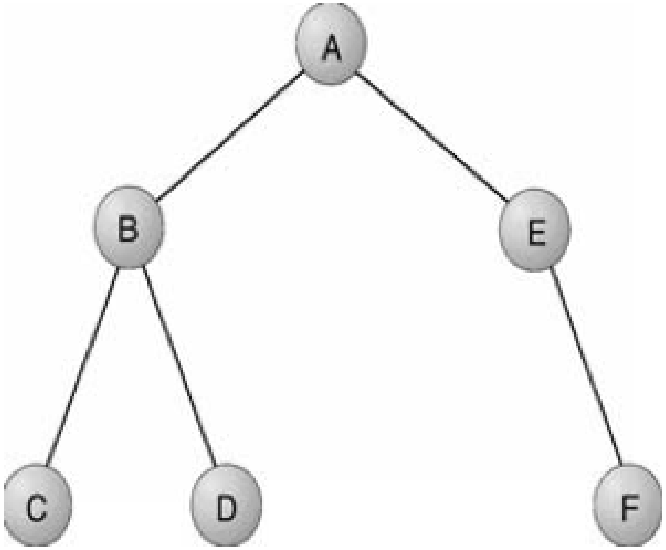
Star

- One **central process** can **communicate** with all **others**
- Other **processes** can only **communicate** with the **central process**.



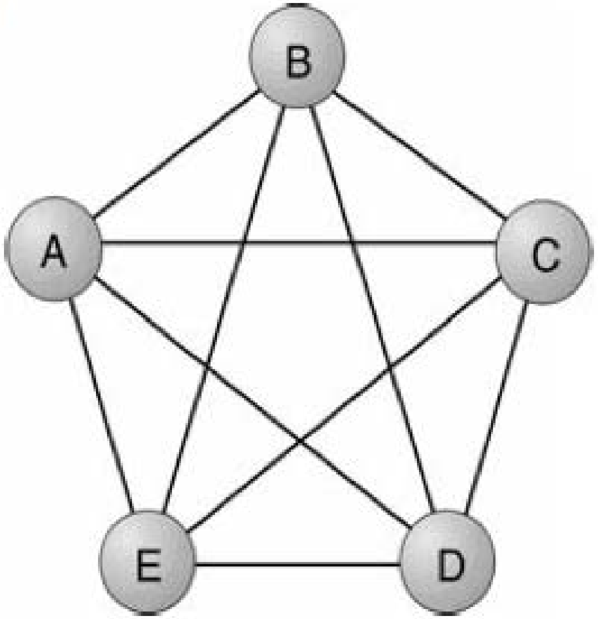
Tree

- **Hierarchical**
- One **process** only **communicates** with its **ancestral** and **descendants**.



Completed connected

- **Every process** communicates with all **others**.



Connection assumptions

- The **algorithms** suppose some or all of the following **properties** for **connection**:

    1. No messages are duplicated
    2. No messages are corrupted
    3. The order of the messages is preserved
        - Messages are received in the same order they are sent
    4. Transmission delay is finite
        - No messages loss
    5. Transmission delay is limited
        - Can be used to detect message loss

Summary

1. Basic Concepts
2. Desirable properties of Distr. Systems
3. Distr. Systems **failures**
4. Design of Distr. Systems
5. The 8 Fallacies
6. Languages for Distr. Programs
7. Logical topologies

Thank you!