# Università degli Studi dell'Aquila

## Academic Year 2017/2018

Course: Distributed Systems (6 CFU, integrated within the NEDAS curriculum with "Web Algorithms" (6 CFU), by Prof. Michele Flammini)

Instructor: Prof. Guido Proietti

| | | |
|---|---|---|
| Schedule: | Tuesday: | 14.30 – 16.15 – Room A1.2 |
| | Thursday: | 14.30 – 16.15 – Room A1.1 |
| Questions?: | Tuesday | 16.30 - 18.30 (or send an email to guido.proietti@univaq.it) |

Slides plus other infos:

http://www.di.univaq.it/~proietti/didattica.html

# Distributed Systems (DS)

In the old days: a number of workstations over a LAN

## Today

Collaborative Computing Systems
- Military command and control
- Massive computation

Distributed Real-time Systems
- Navigation systems
- Airline Traffic Monitoring

Mobile Ad hoc Networks
- Rescue Operations
- Robotics

(Wireless) Sensor Networks
- Habitat monitoring
- Intelligent farming

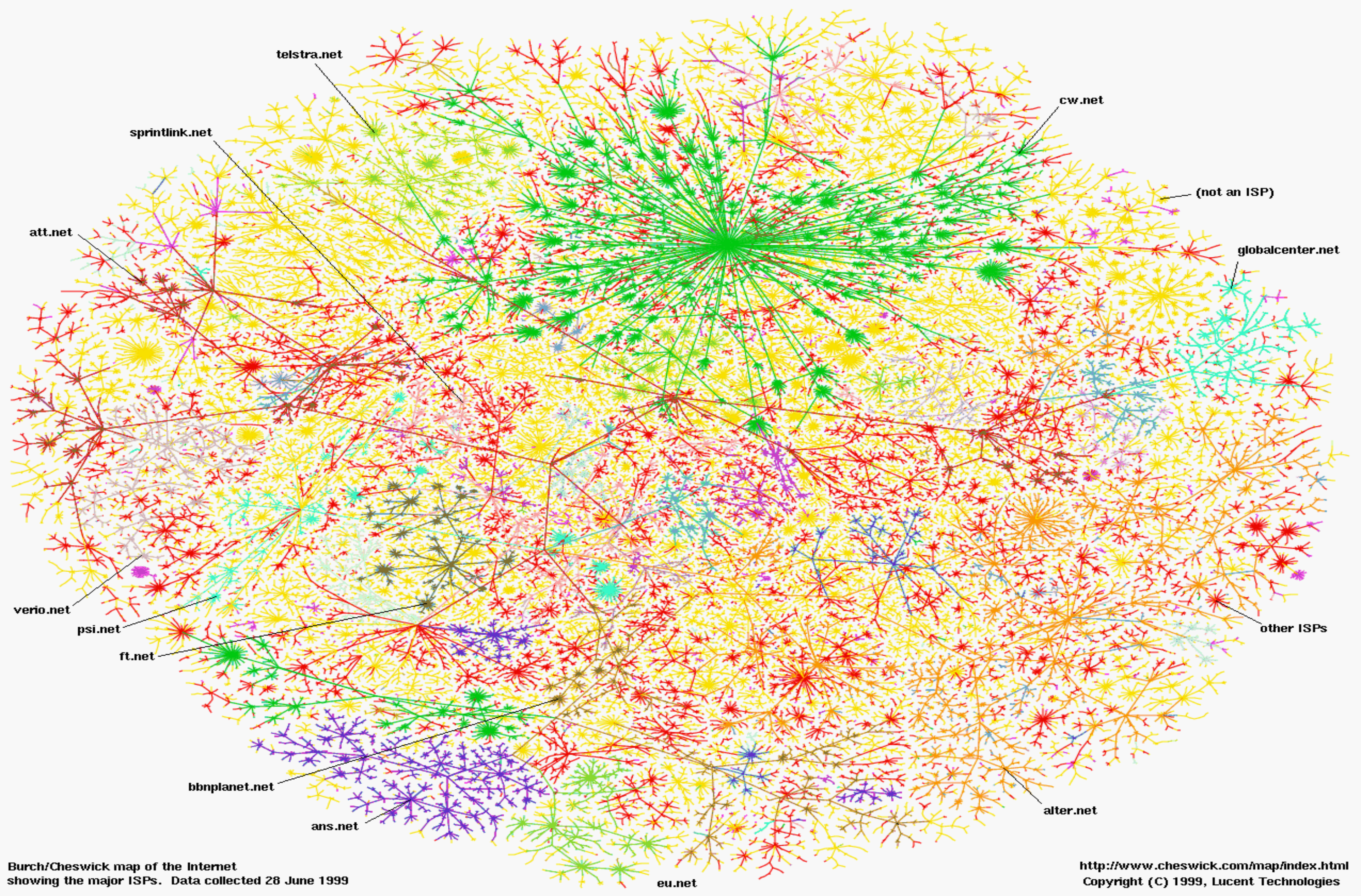
Social Networks

Digital Payment Systems

Grid and Cloud computing

…

# And then, the mother of all DS: the Internet



# Two main ingredients in the course: Distributed Systems + Algorithms

- **Distributed system (DS)**: Broadly speaking, we refer to a set of **autonomous computational devices** (say, **processors**) performing multiple operations/tasks simultaneously, and which influence reciprocally either by **taking actions** or by **exchanging messages** (using an underlying wired/wireless **communication network**), in order to bring the DS to a **final outcome**
- We will be concerned with the **computational aspects** of a DS, namely its ability to coordinate its processors in order to compute/reach a certain outcome/status. As we will see, this will depend on the **behaviour** of its processors:
  - Cooperative: processors are fault-free and cooperate among them and with the system ⇒ Classic field of **distributed computing**
  - Concurrent: processors compete among them for a resource ⇒ Classic field of **synchronization/concurrent computing**
  - Adversarial: processors may fail and operate against the system ⇒ Classic field of **fault-tolerance** in DS
- The emerging field of **game-theoretic aspects of DS**, where processors behave strategically in order to maximize her personal benefit will be studied next year in the class of **Autonomous Networks**

# Two main ingredients in the course: Distributed Systems + Algorithms (2)

- **Algorithm (informal definition)**: effective method, expressed as a finite list of well-defined instructions, for solving a given problem (e.g., calculating a function, implementing a goal, reaching a benefit, etc.)
- The actions performed by each processor in a DS are dictated by a **local algorithm**, and the global behavior of a DS is given by the "composition" (i.e., interaction) of these local algorithms, then named **distributed algorithm**

> **General assumption:** local algorithms are all the same (so-called homogenous setting), otherwise we could force the DS to behave as we want by just mapping different algorithms to different processors, which is unfeasible in reality!

- We will analyze these distributed algorithms in (almost) every respect: existence, correctness, finiteness, efficiency (computational complexity), effectiveness, robustness (w.r.t. to a given fault-tolerance concept), etc.

# Course structure

FIRST PART (12 lectures): Algorithms for COOPERATIVE DS

1. Leader Election
2. Minimum Spanning Tree
3. Maximal Independent Set
4. Minimum Dominating Set

Mid-term **Written** Examination (week 6-10 of November): 10 multiple-choice tests, plus an open-answer question on the above part

SECOND PART (2 lectures): CONCURRENT DS: Mutual exclusion

THIRD PART (6 lectures): Algorithms for UNRELIABLE DS

1. Benign failures: consensus problem
2. Byzantine failures: consensus problem

FOURTH PART (4 lectures): Advanced topics in DS (each student or group of students will read a paper and present it in the classroom)

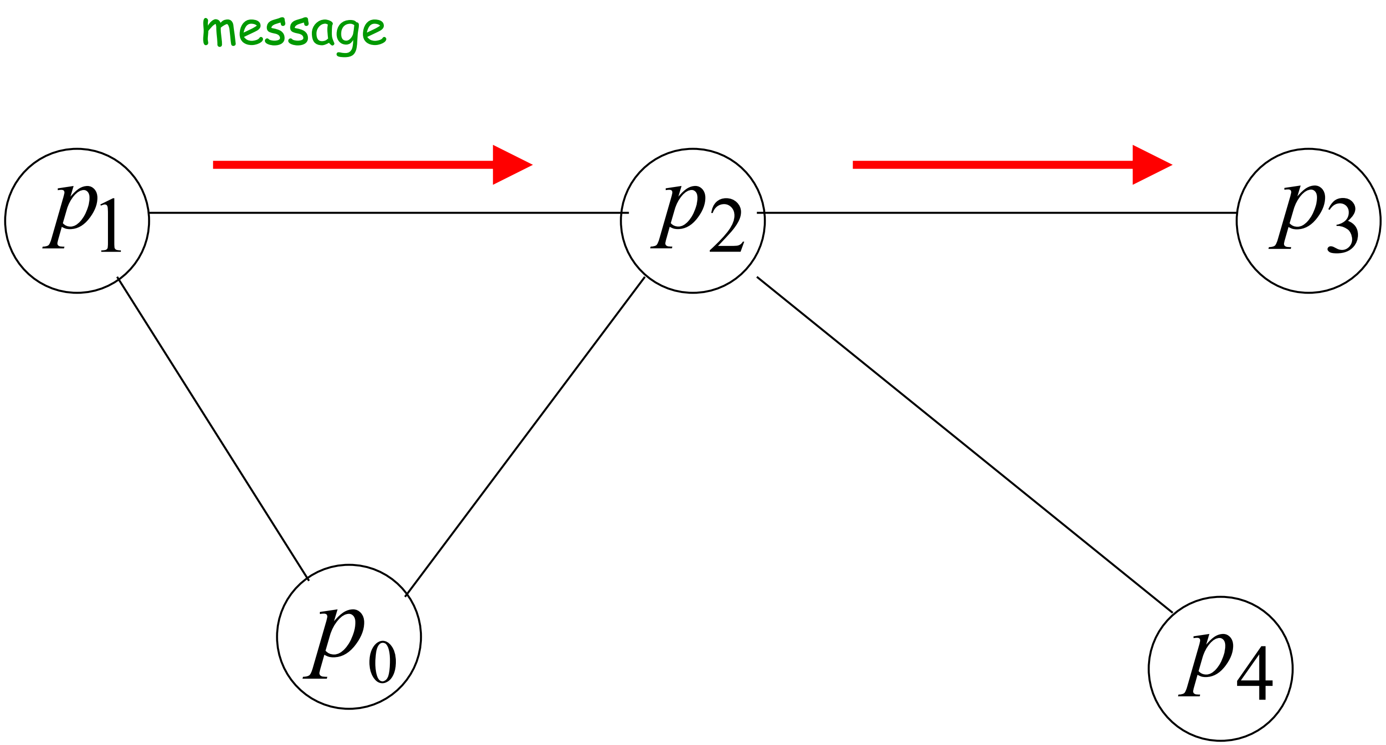
Final **Oral** Examination: this will be concerned with either the whole program or just the second part of it, depending on the outcome of the mid-term exam. There will be fixed a total of 6 dates, namely:

- 3 in January-February
- 2 in June-July
- 1 in September

# The 12 CFU course (Distributed Systems & Web Algorithms)

- For those enrolled in the NEDAS curriculum, there will be a **single** final grade as a result of the grades obtained in this course and in the "Web Algorithms" course (seminars by Prof. Flammini)
- The corresponding exams **can be done separately**, but they must be sustained within the **same calendar year**
- People must register for the 12 CFU exam, but for the actual dates of the 2 courses they must refer to the calendar for the corresponding 6 CFU exams (this is published on the DISIM site, or it can be asked to me and to Prof. Flammini)

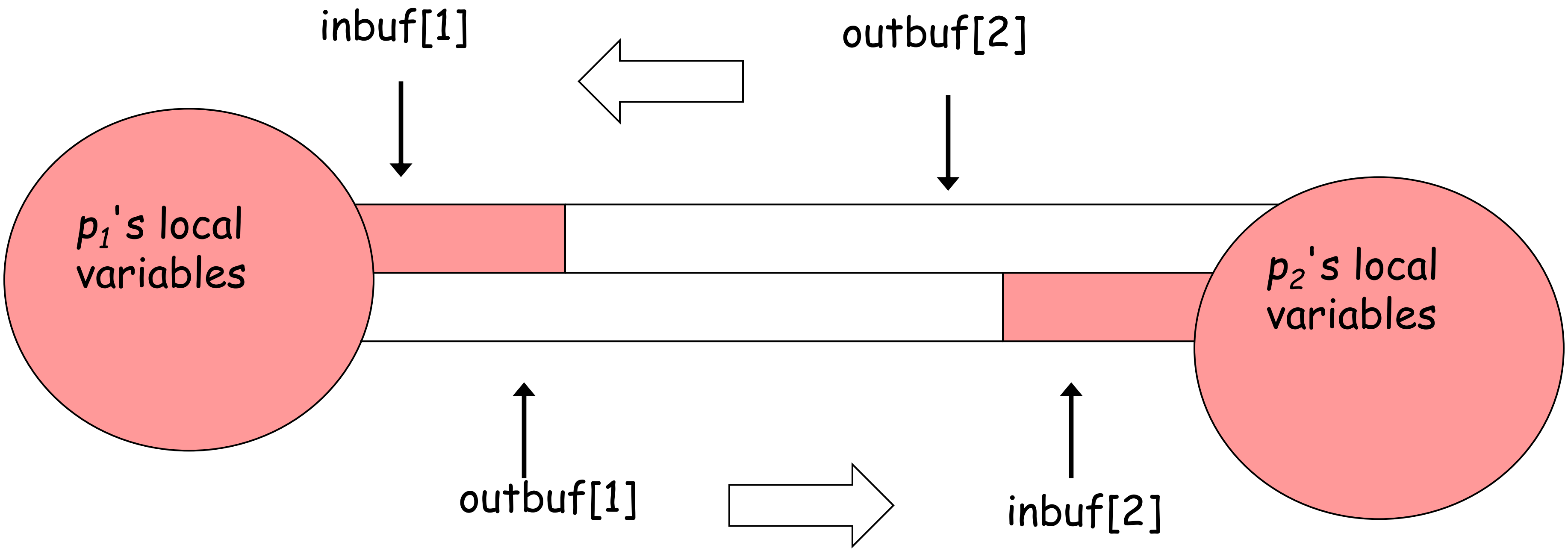
# Cooperative DS: Message Passing Model/System



# The model (MPS)

- n processors $p_0, p_1, \dots, p_{n-1}$ which communicate by exchanging messages
- It can be modelled by a graph $G=(V,E)$: nodes $V$ are the processors, while edges $E$ are the (bidirectional) point-to-point communication channels
- Each processor has a consistent knowledge of its neighbors, numbered from 1 to $r$
- Depending on the context, a processor (or more precisely, its algorithm) may make use of global information about the network, e.g., the size, the topology, etc.
- **Communication** of each processor takes place only through **message exchanges**, using buffers associated with each neighbor, say **outbuf** and **inbuf**
- $Q_i$: the **state set** for $p_i$, containing a distinguished initial state; each state describes the internal status of the processor and the status of the incoming buffers

# Modeling Processors and Channels




Pink area (local vars + inbuf) is the state set of a processor

# Configuration and events

- ✓ **System configuration *C***: *A* vector $[q_0, q_1, \ldots, q_{n-1}]$ where $q_i \in Q_i$ is the state of $p_i$, plus the status of all outbuffers
- ✓ **Events**: Computation events (internal computations plus sending of messages), and message delivering (receipt of messages) events

# Execution

$C_0\ \phi_1\ C_1\ \phi_2\ C_2\ \phi_3$ … where

- ✓ $C_0$ : The initial configuration (all processors are in their initial state and all the buffers are empty)
- ✓ $\phi_i$ : An event
- ✓ $C_i$ : The configuration generated by $\phi_i$ once applied to $C_{i-1}$

# Synchronous MPS

- ✓ Each processor has a (universal) clock, and computation takes place in **rounds**.
- ✓ At each round each processor:
  1. Reads the incoming messages buffer
  2. Makes some internal computations
  3. Sends messages which will be read in the next round.

# Asynchronous MPS

- ✓ No **upper bound** on delivering times
- ✓ **Admissible** asynchronous execution: each message sent is eventually delivered

# Different MPS

- **Topology** of the network (connected undirected graph representing the MPS): clique, ring, star, etc.
- Degree of Synchrony: *asynchronous* versus *synchronous* (**universal clock**)
- Degree of Symmetry: *anonymous* (**processors are indistinguishable**) versus *non-anonymous*: this is a very tricky point, which refers to whether a processor has a distinct ID which can be used during a computation; as we will see, there is a drastic difference in the powerful of a DS, depending on this assumption
- Degree of Uniformity: *uniform* (**number of processors is unknown**) versus *non-uniform*

# Message Complexity

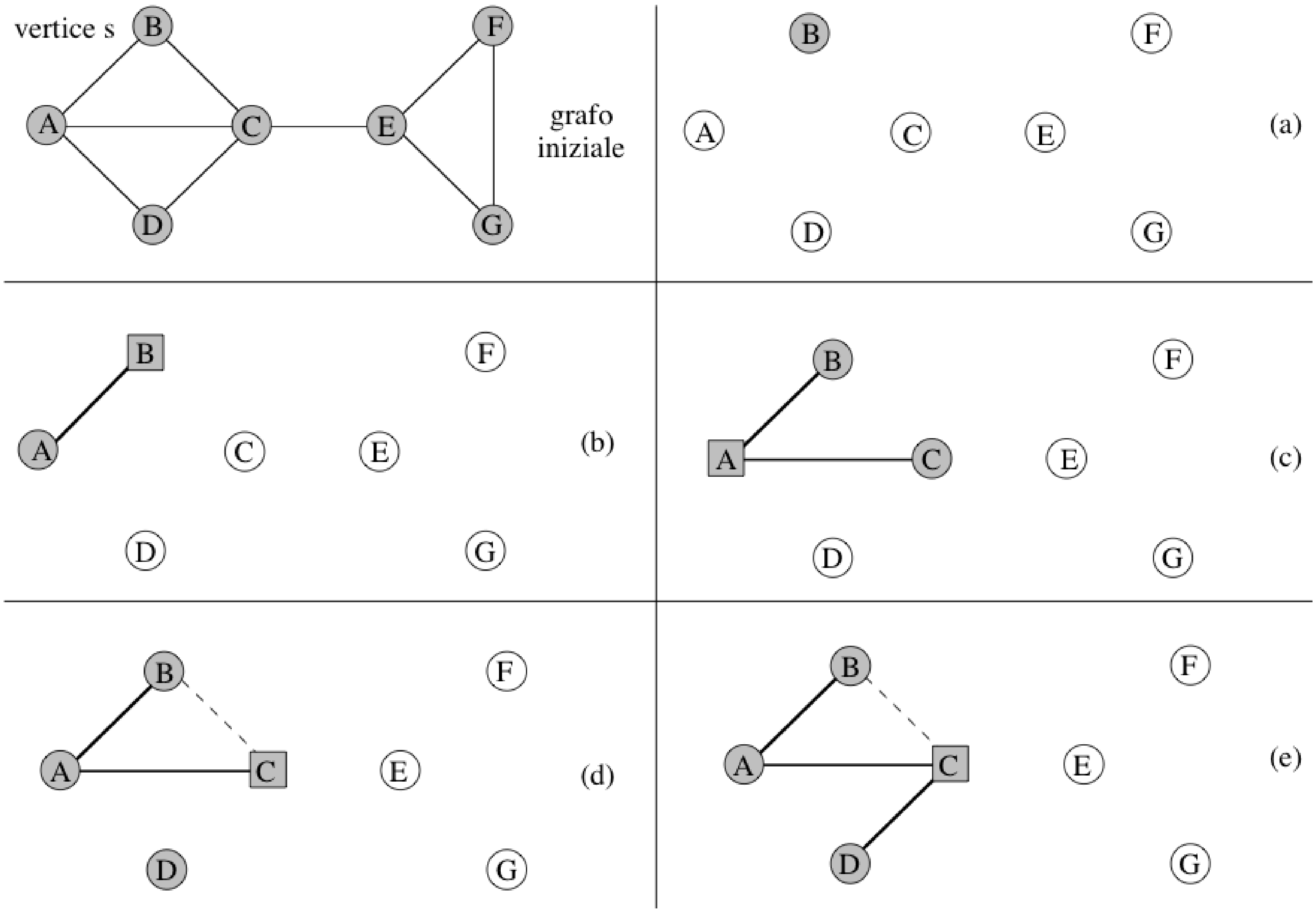
- ✓ We will assume that each message can be **arbitrarily** long
- ✓ According to this model, to establish the efficiency of an algorithm we will only count the **total number** of messages sent during any admissible execution of the algorithm (in other words, the number of *message delivery* events in the worst case), regardless of their size

# Time Complexity

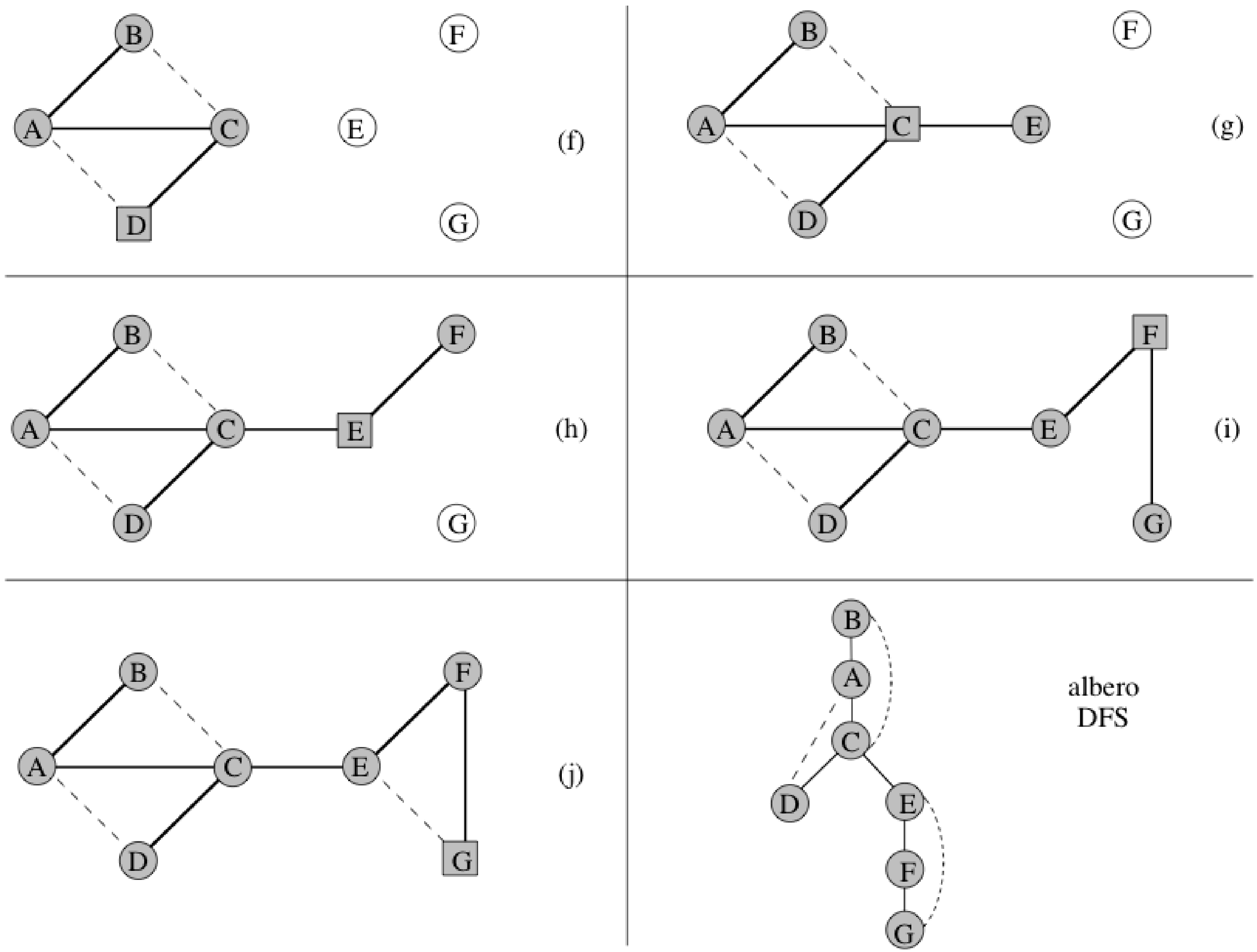
- ✓ We will assume that each processor has **unlimited** computational power
- ✓ According to this model, to establish the efficiency of a synchronous algorithm, we will simply count the number of rounds until termination
- ✓ Asynchronous systems: the time complexity is not really meaningful, since processors do not have a consistent notion of time

- Example: Distributed Depth-First Search visit of a graph
  - Visiting a (connected) graph $G=(V,E)$ means to explore all the nodes and edges of the graph
  - General overview of a sequential algorithm:
    - Begin at some source vertex, $r_0$
    - when a vertex $v$ is visited for the first time
      - » if $v$ has an unvisited neighbor, then visit it and proceed further from it
      - » otherwise, return to parent(v)
    - when we reach the parent of some vertex $v$ such that $parent(v)$ = NULL, then we terminate since $v = r_0$
  - DFS defines a tree, with $r_0$ as the root, which spans all vertices in the graph
    - sequential time complexity = $\Theta(|E|+|V|)$ (we use $\Theta$ notation because every execution of the algorithm costs exactly $|E|+|V|$, in an asymptotic sense)

# DFS: an example (1/2)



# DFS: an example (2/2)



# Distributed DFS: an asynchronous algorithm

- **Distributed version** (token-based): the token traverses the graph in a depth-first manner using the algorithm described above

  1. Start exploration (visit) at a waking-up node (root) r (who wakes-up r? Good question, we will see…)
  2. When v is visited for the first time:
     - 2.1 Inform all its neighbors that it has been visited
     - 2.2 Wait for acknowledgment from all neighbors

       (we will see steps 2.1 and 2.2 are useful in the synchronous case)
     - 2.3 Select an unvisited neighbor node and pass the token to it; if no unvisited neighbor node exists, then pass the token back to the parent node

- Message complexity is $\Theta(|E|)$ (optimal, because of the trivial lower bound of $\Omega(|E|)$ induced by the fact that every node must know the status of each of its neighbors – this requires at least a message for each graph edge)

# Distributed DFS (cont'd.)

Time complexity analysis (synchronous DS)

- Through steps 2.1 and 2.2, we ensure that vertices visited for the first time know which of their neighbors have been visited; this way, each node knows which of its neighbors is still unexplored
- Number of rounds for steps 2.1 and 2.2:
  1. Node v inform all neighbors of it has been visited;
  2. get *Ack* messages from those neighbors;
  3. restart DFS process

$\Rightarrow$ constant number of rounds (i.e., 3) for each new discovered node

- |V| nodes are discovered $\Rightarrow$ time complexity = $\Theta(|V|)$

**Homework**: What does it happen to the algorithm's complexity if we do not inform the neighbors about having been visited?