Advanced and Distributed Algorithms

Academic Year 2023/2024

Nicola Aggio 880008

Index

1 Distributed Computing Environments

The following chapters focus on the algorithmics of distributed computing; that is, on how to solve problems and perform tasks efficiently in a distributed computing environment.
This universe consists of a finite collection of computational entities communicating by means of messages in order to achieve a common goal; for example, to perform a given task, to compute the solution to a problem, to satisfy a request either from the user (i.e., outside the environment) or from other entities. Although each entity is capable of performing computations, it is the collection of all these entities that together will solve the problem or ensure that the task is performed.
In this universe, to solve a problem, we must discover and design a distributed algorithm or protocol for those entities: A set of rules that specify what each entity has to do. The collective but autonomous execution of those rules, possibly without any supervision or synchronization, must enable the entities to perform the desired task to solve the problem. In the design process, we must ensure both **correctness** (i.e., the protocol we design indeed solves the problem) and **efficiency** (i.e., the protocol we design has a "small" cost).

1.1 Entities

The computational unit of a distributed computing environment is called an entity. Depending on the system being modeled by the environment, an entity could correspond to a process, a processor, a switch, an agent, and so forth in the system.

Capabilities Each entity $x \in \mathcal{E}$ is endowed with **local** (i.e., private and non-shared) **memory** M_x. The capabilities of x include:

- **Access** (storage and retrieval) to local memory;
- **Local processing**;
- **Communication** (preparation, transmission, and reception of messages).

Local memory includes a set of defined **registers** whose values are always initially defined, e.g. the **status** register (denoted by *status(x)*), which takes values from a finite set of system states S; the examples of such values are "Idle", "Processing," "Waiting", and so forth. Another example of register is the **input value** register, denoted by *value(x)*.
In addition, each entity $x \in \mathcal{E}$ has available a local alarm **clock** c_x which it can set and reset (turn off).
An entity can perform only four types of **operations**:

- **Local storage** and **processing**;
- **Transmission** of messages;
- Setting of the **alarm clock**;
- **Changing** the **value** of the status register.

External Events The **behavior** of an entity $x \in \mathcal{E}$ is **reactive**: x only responds to external stimuli, which we call external events (or just events); in the absence of stimuli, x is inert and does nothing.
There are three possible external events:

- **Arrival** of a message;

- **Ringing** of the alarm clock;
- **Spontaneous** impulse.

The arrival of a message and the ringing of the alarm clock are the events that are **external** to the entity but originate within the system: the message is sent by another entity, and the alarm clock is set by the entity itself.
Unlike the other two types of events, a spontaneous impulse is triggered by forces external to the system and thus **outside** the universe perceived by the entity. As an example of event generated by forces external to the system, consider an automated banking system: its entities are the bank servers where the data is stored, and the automated teller machine (ATM) machines; the request by a customer for a cash withdrawal (i.e., update of data stored in the system) is a spontaneous impulse for the ATM machine (the entity) where the request is made.

Actions When an external event e occurs, an entity $x \in \mathcal{E}$ will react to e by performing a finite, indivisible, and terminating sequence of operations called **action**.
An action is indivisible (or **atomic**) in the sense that its operations are executed without interruption; in other words, once an action starts, it will not stop until it is finished. An action is terminating in the sense that, once it is started, its execution ends within finite time.
A special action that an entity may take is the **null action**, where the entity does not react to the event.

Behaviour The nature of the action performed by the entity depends on the nature of the event e, as well as on which status the entity is in (i.e., the value of *status(x)*) when the events occur.
Thus the specification will take the form

$$Status \times Event \to Action$$

which will be called a **rule**.
The **behavior** of an entity x is the set $B(x)$ of all the rules that x obeys. This set must be **complete** and **non-ambiguous**: for every possible event e and status value s, there is one and only one rule in $B(x)$ enabled by (s, e). In other words, x must always know exactly what it must do when an event occurs.
The **behavioral specification** of the entire **distributed computing environment** is just the collection of the individual behaviors of the entities. More precisely, the **collective behavior** $B(\mathcal{E})$ of a collection $\mathcal{E}$ of entities is the set

$$B(\mathcal{E}) = B(x) : x \in \mathcal{E}$$

Thus, in an environment with collective behavior $B(\mathcal{E})$, each entity x will be acting (behaving) according to its distributed algorithm and protocol (set of rules) $B(x)$.

Homogeneous Behavior A collective behavior is **homogeneous** if all entities in the system have the same behavior, that is, $\forall x, y \in \mathcal{E}, B(x) = B(y)$.
This means that to specify a homogeneous collective behavior, it is sufficient to specify the behavior of a single entity; in this case, we will indicate the behavior simply by B. An interesting and important fact is the following: **every collective behavior can be made homogeneous**. This means that if we are in a system where different entities have different behaviors, we can write a new set of rules, the same for all of them, which will still make them behave as before.

Example. *Consider a system composed of a network of several identical workstations and a single server; clearly, the set of rules that the server and a workstation obey is not the same as their functionality differs. Still, a single program can be written that will run on both entities without modifying their functionality. We need to add to each entity an input register, my_role, which is initialized to either "workstation" or "server," depending on the entity; for each status–event pair* (s, e) *we create a new rule with the following action:*

$$s \times e \rightarrow \{\textit{if } my\_role = \textit{workstation then } A_{workstation} \textit{ else } A_{server}\}$$

where $A_{workstation}$ *(respectively,* A_{server}*) is the original action associated to* (s, e) *in the set of rules of the workstation (respectively, server). If* (s, e) *did not enable any rule for a workstation (e.g.,* s *was a status defined only for the server), then* $A_{workstation} = NIL$ *in the new rule; analogously for the server.*

It is important to stress that in a **homogeneous system**, although all entities have the **same behavioral description** (software), they **do not have to act in the same way**; their difference will depend solely on the initial value of their input registers. An important consequence of the homogeneous behavior property is that we can concentrate solely on environments where all the entities have the same behavior. From now on, when we mention behavior we will always mean homogeneous collective behavior.

1.2 Communication

In a distributed computing environment, entities communicate by transmitting and receiving messages. The message is the unit of **communication** of a distributed environment.
An entity communicates by **transmitting messages** to and **receiving messages** from other entities. The **set of entities** with which an entity can communicate directly is not necessarily $\mathcal{E}$; in other words, it is possible that an entity can communicate directly only with a subset of the other entities. We denote by $N_{out}(x) \subseteq \mathcal{E}$ the set of entities to which x can transmit a message directly; we shall call them the **out-neighbors** of x. Similarly, we denote by $N_{in}(x) \subseteq \mathcal{E}$ the set of entities from which x can receive a message directly; we shall call them the **in-neighbors** of x.
The **neighborhood** relationship defines a directed graph $G = (V, E)$ where V is the set of vertices and $E \subseteq V \times V$ is the set of edges; the vertices correspond to entities, and $(x, y) \in E$ if and only if the entity (corresponding to) y is an out-neighbor of the entity (corresponding to) x.
In summary, an **entity** can only **receive** messages from its **in-neighbors** and **send** messages to its **out-neighbors**. Messages received at an entity are processed there in the order they arrive; if more than one message arrive at the same time, they will be processed in arbitrary order. Entities and communication may fail.

1.3 Axioms and restrictions

The definition of distributed computing environment with point-to-point communication has two basic axioms, one on communication delay, and the other on the local orientation of the entities in the system.

1.3.1 Axioms

Communication Delays Communication of a message involves many activities: preparation, transmission, reception, and processing. In real systems described by our model, the **time** required by these activities is **unpredictable**. For example, in a communication network a

message will be subject to queueing and processing delays, which change depending on the network traffic at that time; for example, consider the delay in accessing (i.e., sending a message to and getting a reply from) a popular web site.
The totality of delays encountered by a message will be called the **communication delay** of that message. The axiom states that, in the **absence of failures**, **communication delays are finite**. In other words, in the absence of failures, a message sent to an out-neighbor will eventually arrive in its integrity and be processed there.
Note that the Finite Communication Delays axiom does not imply the existence of any bound on transmission, queueing, or processing delays; it only states that in the absence of failure, a message will arrive after a finite amount of time without corruption.

Local Orientation An entity can communicate directly with a subset of the other entities: its neighbors. The only other axiom in the model is that an **entity can distinguish between its neighbors**, in particular it can distinguish among its in-neighbors and among its out-neighbors. An entity is capable of sending a message only to a specific out-neighbor (without having to send it also to all other out-neighbors). Also, when processing a message (i.e., executing the rule enabled by the reception of that message), an entity can distinguish which of its in-neighbors sent that message.
In other words, each entity x has a local function λ_x associating labels, also called **port numbers**, to its incident links (or **ports**), and this function is injective. We denote port numbers by $\lambda_x(x, y)$, the label associated by x to the link (x, y). Let us stress that this label is local to x and in general has no relationship at all with what y might call this link (or x, or itself). Note that for each edge $(x, y) \in \mathcal{E}$, there are two labels: $\lambda_x(x, y)$ local to x and $\lambda_y(x, y)$ local to y.

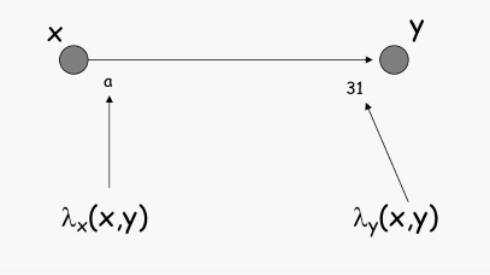


Figure 1: *Every edge has two labels.*

Because of this axiom, we will always deal with edge-labeled graphs (G, λ), where $\lambda = \lambda_x : x \in V$ is the set of these injective labelings.

1.3.2 Restrictions

In general, a distributed computing system might have additional **properties** or capabilities that can be exploited to solve a problem, to achieve a task, and to provide a service. This can be achieved by using these properties and capabilities in the set of rules.
However, any property used in the protocol limits the applicability of the protocol. In other words, any additional property or capability of the system is actually a **restriction** (or submodel) of the general model.
The restrictions can be varied in nature and type: they might be related to communication properties, reliability, synchrony, and so forth. In the following section, we will discuss some of the most common restrictions.

Communication Restrictions The first category of restrictions includes those relating to **communication** among **entities**.
Queueing policy A **link** (x, y) can be viewed as a **channel** or a **queue**: x sending a message to y is equivalent to x inserting the message in the channel. In general, all kinds of situations are possible; for example, messages in the channel might overtake each other, and a later message might be received first. Different restrictions on the model will describe different disciplines employed to manage the channel; for example, first-in-first-out (**FIFO**) queues are characterized by the following restriction.

- **Message Ordering**: In the **absence of failure**, the **messages** transmitted by an entity to the same out-neighbor will arrive in the **same order** they are **sent**.

Note that Message Ordering **does not imply** the **existence** of any **ordering** for messages transmitted to the same entity **from different edges**, nor for messages sent by the same entity on different edges.
Link property Entities in a communication system are connected by **physical links**, which may be very different in capabilities. The examples are **simplex** and **full-duplex** links. With a fully duplex line it is possible to transmit in both directions. Simplex lines are already defined within the general model. A duplex line can obviously be described as two simplex lines, one in each direction; thus, a system where all lines are fully duplex can be described by the following restriction:

- **Reciprocal communication**: $\forall x \in \mathcal{E}, N_{in}(x) = N_{out}(x)$. In other words, if $(x, y) \in E$ then also $(y, x) \in E$.

Notice that, however, $(x, y) \neq (y, x)$, and in general $\lambda_x(x, y) \neq \lambda_x(y, x)$; furthermore, x might not know that these two links are connections to and from the same entity. A system with fully duplex links that offers such a knowledge is defined by the following restriction.

- **Bidirectional links**: $\forall x \in \mathcal{E}, N_{in}(x) = N_{out}(x)$ and $\lambda_x(x, y) = \lambda_x(y, x)$.

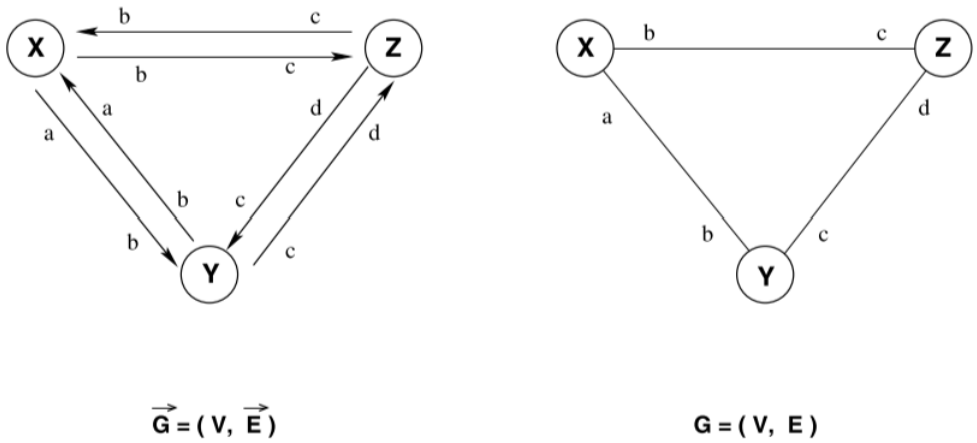


Figure 2: *In a network with bidirectional links we consider the corresponding undirected graph.*

Reliability restrictions Other types of restrictions are those related to reliability, faults, and their detection.
Detection of Faults Some systems might provide a reliable fault-detection mechanism. Following are two restrictions that describe systems that offer such capabilities in regard to component failures:

- **Edge failure detection**: $\forall (x, y) \in E$, both x and y will detect whether (x, y) has failed and, following its failure, whether it has been reactivated;
- **Entity failure detection**: $\forall x \in V$, all in- and out-neighbors of x can detect whether x has failed and, following its failure, whether it has recovered.

Restricted Types of Faults In some systems only some types of failures can occur: for example, messages can be lost but not corrupted. Each situation will give rise to a corresponding restriction. More general restrictions will describe systems or situations where there will be no failures:

- **Guaranteed delivery**: **Any message** that is sent will be **received** with its content **uncorrupted**. Under this restriction, protocols do not need to take into account omissions or corruptions of messages during transmission. Even more general is the following;
- **Partial reliability**: **No failures will occur**. Under this restriction, protocols do not need to take failures into account. Note that under Partial Reliability, failures might have occurred before the execution of a computation. A totally fault-free system is defined by the following restriction;
- **Total reliability**: Neither have **any failures occurred** nor **will they occur**.

Clearly, protocols developed under this restriction are not guaranteed to work correctly if faults occur.

Topological Restrictions In general, an entity is not directly connected to all other entities; it might still be able to communicate information to a remote entity, using others as relayer. A system that provides this capability for all entities is characterized by the following restriction:

- **Connectivity**: The communication topology G is **strongly connected**.

That is, from every vertex in G it is possible to reach every other vertex. In case the restriction "Bidirectional Links" holds as well, connectedness will simply state that G is **connected**.

Time Restrictions An interesting type of restrictions is the one relating to time. In fact, the general model makes no assumption about delays (except that they are finite).

- **Bounded communication delays**: There exists a constant Δ such that, in the absence of failures, the communication delay of any message on any link is at most Δ.

A special case of bounded delays is the following:

- **Unitary communication delays**: In the absence of failures, the communication delay of any message on any link is one unit of time. The general model also makes no assumptions about the local clocks;
- **Synchronized clocks**: All local clocks are incremented by one unit simultaneously and the interval of time between successive increments is constant.

Usually, we'll consider asynchronous systems.

1.4 Cost and complexity

We will use two types of measures: the **amount of communication activities** and the **time** required by the execution of a computation. They can be seen as measuring costs from the system point of view (*How much traffic will this computation generate and how busy will the system be?*) and from the user point of view (*How long will it take before I get the results of the computation?*)

1.4.1 Amount of communication activities

The transmission of a message through an out-port (i.e., to an out-neighbor) is the basic communication activity in the system; note that the transmission of a message that will not be received because of failure still constitutes a communication activity. Thus, to measure the amount of communication activities, the most common function used is the number of message transmissions M, also called **message cost**. So in general, given a protocol, we will measure its communication costs in terms of the **number of transmitted messages**.

1.4.2 Time

An important measure of efficiency and complexity is the **total execution delay**, that is, the delay between the time the first entity starts the execution of a computation and the time the last entity terminates its execution.
In the general model there is **no assumption** about time except that communication delays for a single message are finite in absence of failure. In other words, **communication delays** are in general **unpredictable**. Thus, even in the absence of failures, the total execution delay for a computation is totally unpredictable; furthermore, two distinct executions of the same protocol might experience drastically different delays. In other words, we cannot accurately measure time. We, however, can measure time assuming particular conditions. The measure usually employed is the **ideal execution delay** or ideal time complexity, T: the execution delay experienced under the restrictions "Unitary Transmission Delays" and "Synchronized Clocks;" that is, when the system is synchronous and (in the absence of failure) takes one unit of time for a message to arrive and to be processed.
A very different cost measure is the **causal time complexity**, T_{causal}. It is defined as the length of the longest chain of causally related message transmissions, over all possible executions. Causal time is seldom used and is very difficult to measure exactly; we will employ it only once, when dealing with synchronous computations.

1.5 Broadcasting

Let us clarify the concepts expressed so far by means of an example. Consider a distributed computing system where one entity has some important information unknown to the others and would like to share it with everybody else. This problem is called **broadcasting**.
Let $\mathcal{E}$ be the **collection of entities** and G be the **communication topology**. To simplify the discussion, we will make some additional **assumptions** (i.e., restrictions) on the system:

1. **Bidirectional links**; that is, we consider the undirected graph G;
2. **Total reliability**, that is, we do not have to worry about failures. Observe that, if G is disconnected, some entities can never receive the information, and the broadcasting problem will be unsolvable. Thus, a restriction that (unlike the previous two) we need to make is as follows.

3. **Connectivity**; that is, G is connected. Further observe that built in the definition of the problem, there is the assumption that only the entity with the initial information will start the broadcast. Thus, a restriction built in the definition is as follows.
4. **Unique Initiator**, that is, only one entity will start.

1.5.1 Flooding algorithm

A simple strategy for solving the broadcast problem is the following: *"if an entity knows the information, it will share it with its neighbors."*
To construct the set of rules implementing this strategy, we need to define the set S of **status values**; from the statement of the problem it is clear that we need to distinguish between the entity that initially has the information and the others: $\{\text{initiator}, \text{sleeping}\} \subseteq S$. The process can be started only by the initiator, while the other entities are sleeping; let I denote the information to be broadcasted.
Here is the **set of rules** $B(x)$ (the same for all entities).

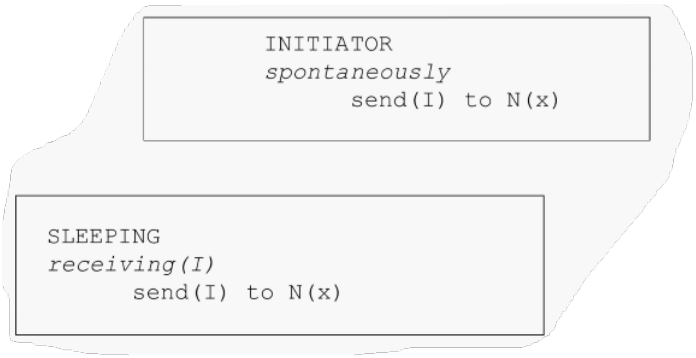


Figure 3: *Set of rules of the Flood algorithm*

Because of **connectivity** and **total reliability**, every entity will eventually receive the information. Hence, the protocol achieves its goal and **solves the broadcasting problem**.
However, there is a serious problem with these rules: the **activities** generated by the protocol **never terminate**.
Consider, for example, the simple system with three entities x, y, z connected to each other. Let x be the initiator, y and z be idle, and all messages travel at the same speed; then y and z will be forever sending messages to each other (as well as to x).
To avoid this unwelcome effect, an **entity** should **send** the **information** to its neighbors **only once**: the first time it acquires the information. This can be achieved by introducing a new status done; that is $S = \{\text{initiator}, \text{sleeping}, \text{done}\}$.
This time the communication activities of the protocol **terminate**: Within finite time all entities become done; since a done entity knows the information, the **protocol** is **correct**. Note that depending on transmission delays, different executions are possible; one such execution in an environment composed of three entities x, y, z connected to each other.
Notice that in this case:

- The **states** are INITIATOR, SLEEPING and DONE;
- The **events** are SPONTANEOUSLY and RECEIVING(I);
- The **actions** are $send(I)$, etc..

Note that **entities terminate** their execution of the protocol (i.e., become done) at **different times**; it is actually possible that an entity has terminated while others have not yet started.

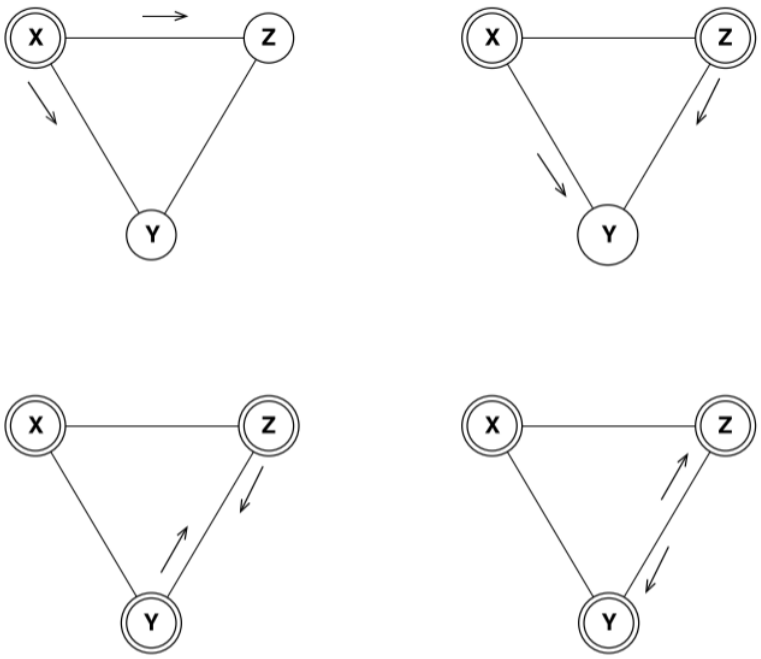


Figure 4: *Execution of the Flood algorithm.*

```
INITIATOR
spontaneously
      send(I) to N(x)
      become(DONE)

SLEEPING
receiving(I)
      send(I) to N(x) - {sender}
      become(DONE)
```

Figure 5: *Set of rules of the Flood algorithm.*

This is something very typical of distributed computations: There is a difference between local termination and global termination.
Notice also that in this protocol **nobody** ever **knows** when the **entire process is over**. We will examine these issues in details in other chapters, in particular when discussing the problem of termination detection.
Now, our next goal is to prove the **correctness** of the algorithm, in particular:

1. The algorithm **correctly solves the problem**: if G is connected and there is total reliability, every entity will eventually receive the information;
2. The algorithm **terminates**: if an entity has received the message it will enter the state done and it will terminate.

First of all, let us determine the **number of message transmissions**. Each entity, whether initiator or not, sends the information to all its neighbors. Hence the total number of messages transmitted is exactly:

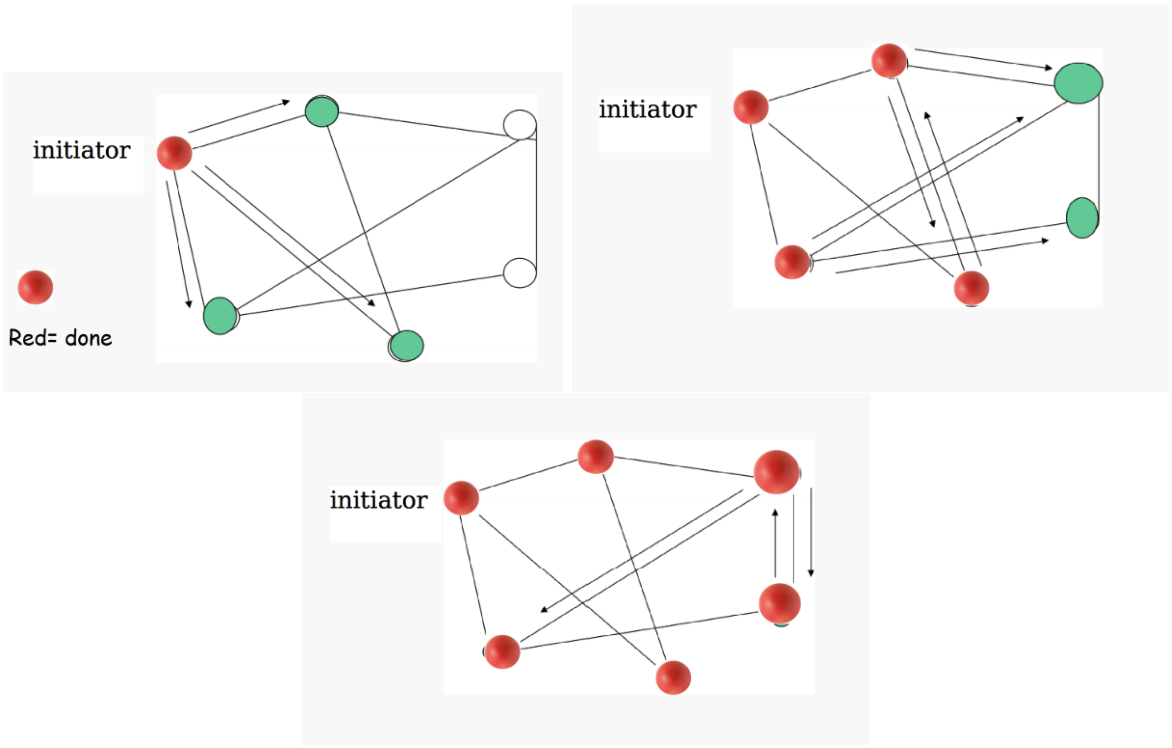


Figure 6: *Example of execution of the Flood algorithm.*

$$\mathbf{M}[\text{Flooding}] = \sum_{x \in \mathcal{E}} |N(x)| = 2|E| = 2m$$

We can actually **reduce the cost**. Currently, when an idle entity receives the message, it will broadcast the information to all its neighbors, including the entity from which it had received the information; this is clearly **unnecessary**. Recall that, by the Local Orientation axiom, an entity can distinguish among its neighbors; in particular, when processing a message, it can identify from which port it was received and avoid sending a message there. In this case the number of messages will be:

$$\mathbf{M}[\text{Flooding}] = \sum_{x \in \mathcal{E}} |N(x)| - \sum_{x \neq s} 1 = 2m - (n-1)$$

Let us examine now the **ideal time complexity of flooding**. Let $d(x, y)$ denote the **distance** (i.e., the length of the shortest path) between x and y in G. Clearly the message sent by the initiator has to reach every entity in the system, including the furthermost one from the initiator. So, if x is the initiator, the **ideal time complexity** will be $r(x) = \max d(x, y) : y \in \mathcal{E}$, which is called the eccentricity (or **radius**) of x. In other words, the total time depends on which entity is the initiator and thus cannot be known precisely beforehand. We can, however, determine exactly the ideal time complexity in the worst case.
Since any entity could be the initiator, the ideal time complexity in the **worst case** will be $d(G) = \max r(x) : x \in \mathcal{E}$, which is the **diameter** of G. In other words, the ideal time complexity will be at most the diameter of G:

$$\mathbf{T}[\text{Flooding}] \leq d(G)$$

2 Basic problems and protocols

The aim of this chapter is to introduce some of the basic, primitive, computational problems and solution techniques. These problems are basic in the sense that their solution is commonly (sometimes frequently) required for the functioning of the system (e.g., *broadcast* and *wake-up*); they are primitive in the sense that their computation is often a preliminary step or a module of complex computations and protocols (e.g., traversal and spanning-tree construction).

2.1 Broadcast

Consider a distributed computing system where only one entity, x, knows some important information; this entity would like to share this information with all the other entities in the system. This problem is called **broadcasting** (*Bcast*), and already we have started its examination in the previous chapter.
To solve this problem means to design a set of rules that, when executed by the entities, will lead (within finite time) to a configuration where all entities will know the information; the solution must work regardless of which entity has the information at the beginning. Built-in the definition of the problem, there is the assumption, **Unique Initiator** (UI), that only one entity will start the task. Actually, this assumption is further restricted, because the unique initiator must be the **one with the initial information**; we shall denote this restriction by $UI+$. To solve this problem, every entity must clearly be involved in the computation.
Hence, for its solution, broadcasting requires the **Connectivity** (CN) restriction (i.e., every entity must be reachable from every other entity) otherwise some entities will never receive the information. We have seen a simple solution to this problem, *Flooding*, under two additional restrictions: **Total Reliability** (TR) and **Bidirectional Links** (BL). Recall that the set $R = \{BL, CN, TR\}$ is the set of **standard restrictions**.

2.1.1 Cost of broadcasting

As we have seen, the solution protocol *Flooding* uses $O(m)$ messages and, in the worst case, $O(d)$ ideal time units, where d is the diameter of the network.
The first and natural question is whether these costs could be reduced significantly (i.e., in order of magnitude) using a **different approach** or technique, and if so, by how much. This question is equivalent to ask what is the complexity of the broadcasting problem.
To answer this type of questions we need to establish a **lower bound**: to find a bound f (typically, a function of the size of the network) and to prove that the cost of every solution algorithm is at least f.
We will denote by **M**[Bcast/RI+] and **T**[Bcast/RI+] the **message** and the **time complexity** of broadcasting under $RI+ = R \cup UI+$, respectively.

Time complexity A **lower bound** on the amount of ideal time units required to perform a broadcast is simple to derive: **Every entity must receive** the **information** regardless of how distant they are from the initiator, and any entity could be the initiator. Hence, in the worst case,

$$\mathbf{T}[\text{Bcast/RI+}] \geq \max d(x, y) : x, y \in V = d$$

The fact that *Flooding* performs the broadcast in d ideal time units means that the **lower bound is tight** (i.e., it can be achieved) and that *Flooding* is **time optimal**. In other words, we know exactly the ideal time complexity of broadcasting: the **ideal time complexity** of broadcasting under $RI+$ is $\Theta(d)$.

Message complexity Let us now consider the **message complexity**. An obvious lower bound on the number of messages is also easy to derive: in the end, **every entity** must know the **information**; thus a message must be received by each of the $n-1$ entities (excluded himself), which initially did not have the information. Hence,

$$\mathbf{M}[\text{Bcast/RI+}] \geq n-1$$

With a little extra effort, we can derive a more accurate lower bound:

$$\mathbf{M}[\text{Bcast/RI+}] \geq m$$

This means that any broadcasting algorithm requires $\Omega(m)$ messages. Since *Flooding* solves broadcasting with $2m-n+1$ messages, this implies $\mathbf{M}[\text{Bcast/RI+}] \geq 2m-n+1$. Since the upper bound and the lower bound are of the same order of magnitude, we can summarize: the **message complexity** of broadcasting under $RI+$ is $\Theta(m)$.
The immediate consequence is that, in order of magnitude, *Flooding* is a **message-optimal solution**. Thus, if we want to design a new protocol to improve the $2m-n+1$ cost of *Flooding*, the best we can hope to achieve is to reduce the constant 2; in any case, since $\mathbf{M}[\text{Bcast/RI+}] \geq m$, the reduction cannot bring the constant below.

2.1.2 Broadcasting in special networks

The results we have obtained so far apply to generic solutions; that is, solutions that do not depend on G and can thus be applied **regardless of the communication topology** (provided it is undirected and connected).
Next, we will consider performing the broadcast in **special networks**. Throughout we will assume the standard restrictions plus $UI+$.

Broadcasting in Trees Consider the case when G is a **tree**; that is, G is connected and contains no cycles. In a tree, $m = n-1$; hence, the use of protocol *Flooding* for broadcasting in a tree will cost $2m-(n-1) = 2(n-1)-(n-1) = n-1$ messages, i.e. it is **optimal**.
Note that this cost is achieved even if the entities do not know that the network is a tree, and an interesting side effect of broadcasting on a tree is that the tree becomes rooted in the initiator of the broadcast.
Finally, *Flooding* works very **well** on **trees**, so the idea to solve the broadcasting problem is the following:

1. Build a **spanning tree** of G;
2. Run the *Flooding* algorithm.

However, building a spanning tree is costly.

Broadcasting in Oriented Hypercubes A communication topology that is commonly used as an interconnection network is the (k-dimensional) labeled **hypercube**, denoted by H_k.
A oriented hypercube H_1 of dimension $k=1$ is just a pair of nodes called (in binary) "0" and "1", connected by a link labeled "1" at both nodes.
In a hypercube H_k of dimension $k>1$, every node has a label of k bits, and the arc label represents the position of the bit, starting from the right, where the node labels differ (i.e. the Hamming distance). Notice that each node does not know its own label, but only the ones of the neighboring nodes.

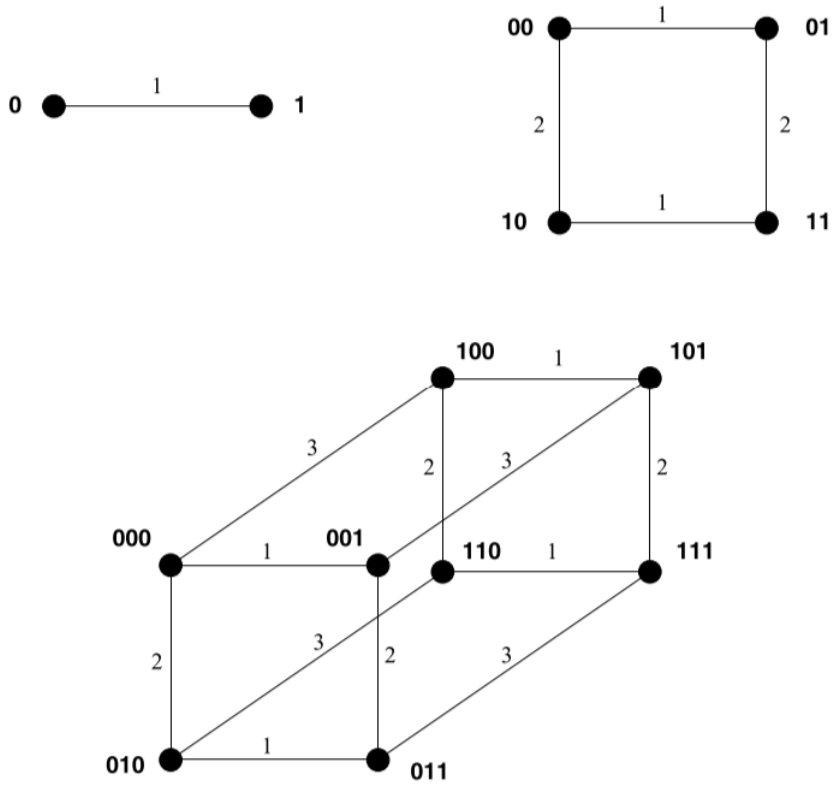


Figure 7: *Oriented Hypercubes.*

A hypercube of dimension k has $n = 2^k$ nodes; each node has k links, labeled $1, 2, .., k$. Hence, the total number of links is $m = nk/2 = (n/2) \log n = O(n \log n)$.
A straightforward application of *Flooding* in a hypercube will cost $2m-(n-1) = n \log n-(n-1) = n \log n/2 + 1 = O(n \log n)$ messages. However, hypercubes are highly structured networks with many interesting **properties**. We can exploit these special properties to construct a more efficient broadcast. Obviously, if we do so, the protocol cannot be used in other networks.
Consider the following simple strategy, called *HyperFlood*:

1. The **initiator sends** the message to all its **neighbors**;
2. A **node receiving** a message from the link labeled l will **send** the messages only to those **neighbors** with label $l' < l$.

The only difference between *HyperFlood* and the normal *Flooding* is in step 2: **Instead** of sending the message to **all neighbors except the sender**, the entity will forward it **only** to **some of them**, which will depend on the label of the port from where the message is received. As we will see, this strategy correctly performs the broadcast using only $n - 1$ messages (instead of $O(n \log n)$), which is **optimal**.
Let us first examine **termination** and **correctness**. Let $H_k(x)$ denote the subgraph of H_k induced by the links where messages are sent by *HyperFlood* when x is the initiator. Clearly every node in $H_k(x)$ will receive the information.

Theorem. *HyperFlood correctly* ***terminates****.*

Also as an exercise it is left the proof that for every x, the eccentricity of x in $H_k(x)$ is k; this

implies that the **optimal** time delay of *HyperFlood* in H_k is always k. That is,

$$\mathbf{T}[\text{HyperFlood}/H_k] = k$$

These costs are the best that any broadcast algorithm can perform in a hypercube regardless of how much more knowledge they have.
Recalling that the distance among two entities in the hypercube is given by the Hamming distance, then the broadcast time, which is equal to k, represents the number k of bits used by the labels, and the diameter of the graph.

Broadcasting in Complete Graphs Among all network topologies, the **complete graph** is the one with the most links: Every entity is connected to all others; thus $m = n(n-1)/2 = O(n^2)$ (recall we are considering bidirectional links), and $d = 1$.
The use of a generic protocol will require $O(n^2)$ messages. But this is really unnecessary.
Broadcasting in a complete graph is easily accomplished: Because everybody is connected to everybody else, the **initiator** just needs to **send** the information to its **neighbors** (i.e., execute the command "*send*(I) to $N(x)$") and the broadcast is completed. This uses only $n-1$ messages and $d = 1$ ideal time.
Clearly this protocol, *KBcast*, works only in a complete graph, that is under the additional restriction that G is a complete graph.
Summarizing, the message and the ideal time complexity of broadcasting in a complete graph under $RI+$ is $\Theta(k)$ are

$$\mathbf{M}[\text{Bcast}/\text{RI+;K}] = n - 1$$

and

$$\mathbf{T}[\text{Bcast}/\text{RI+;K}] = 1$$

respectively.

2.2 Spanning Tree construction

In a distributed computing environment, to construct a spanning tree of G means to move the system from an initial system configuration, where each entity is just aware of its own neigbors, to a system configuration where

1. Each **entity** x has selected a **subset** Tree-neighbors$(x) \subseteq N(x)$ and
2. The **collection** of all the corresponding **links** forms a **spanning tree** of G.

In the following we will indicate $T(G)$ simply by T, if no ambiguity arises.
Note that T is not known a priori to the entities and might not be known after it has been constructed: an entity needs to know only which of its neighbors are also its neighbors in the spanning tree T. As before, we will restrict ourselves to connected networks with bidirectional links and further assume that no failure will occur.
We will first assume that the construction will be started by only one entity (i.e., **Unique Initiator** (UI) restriction); that is, we will consider spanning-tree construction under restrictions RI. We will then consider the general problem when **any number of entities** can independently **start** the construction. As we will see, the situation changes dramatically from the single-initiator scenario.

2.2.1 SPT construction with a single initiator: Shout

Consider the entities; they do not know G, not even its size. The only things an entity is aware of are the **labels** on the ports leading to its neighbors (because of the Local Orientation axiom) and the fact that, if it sends a message to a neighbor, the message will eventually be **received** (because of the Finite Communication Delays axiom and the Total Reliability restriction). How, using just this information, can a spanning tree be constructed?
The answer is surprisingly simple. Each entity needs to know which of its neighbors are also neighbors in the spanning tree. The solution strategy is just **"ask"**.

Strategy Ask-Your-Neighbors

1. The **initiator** s will "ask" its neighbors; that is, it will send a message Q = *("Are you my neighbor in the spanning tree"?)* to all its neighbors;
2. An entity $x \neq s$ will reply *"Yes"* only the **first time** it is asked and, in this occasion, it will **ask all its other neighbors**; otherwise, it will reply *"No"*. The initiator s will always reply *"No"*;
3. Each entity **terminates** when it has **received** a **reply from all neighbors** to which it asked the question.

For an entity x, its neighbors in the spanning tree T are the neighbors that have replied *"Yes"* and, if $x \neq s$, also the neighbor from which the question was first asked.
The corresponding set of rules is depicted in the image below where in bold are shown the tree links and in dotted lines the non-tree links.

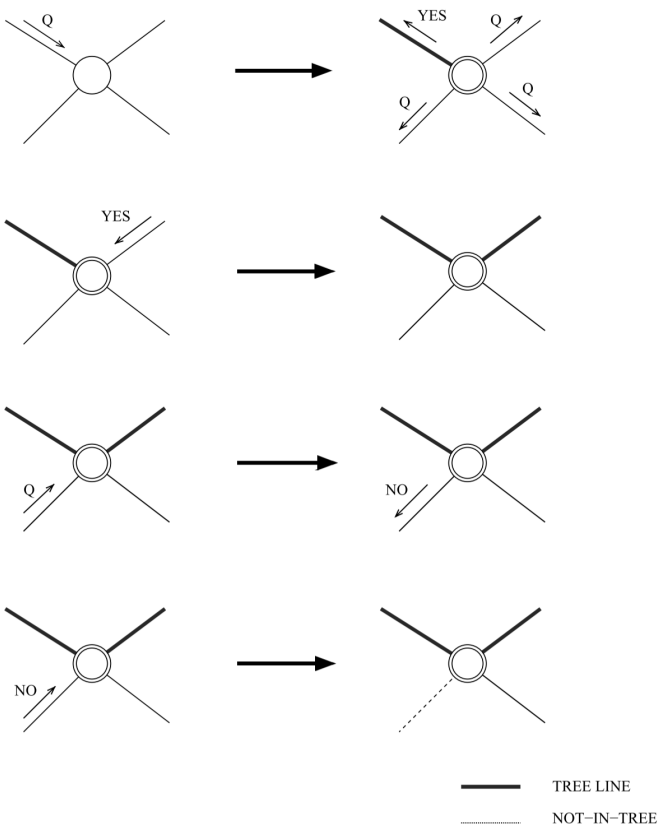


Figure 8: *Set of rules of Shout.*

In general, we can see that each time a Q is sent, we need to receive the answer from all the neighbors, in order to know whether the edges belong to the spanning tree or not. The protocol *Shout* implementing this strategy is shown below. Initially, all nodes are in status *idle* except the sole initiator.

PROTOCOL Shout

- Status: $\mathcal{S}$ = {INITIATOR,IDLE,ACTIVE,DONE};
  $\mathcal{S}_{INIT}$ = {INITIATOR,IDLE};
  $\mathcal{S}_{TERM}$ = {DONE}.
- Restrictions: **R** ;UI.

```
INITIATOR
        Spontaneously
        begin
            root:= true;
            Tree-neighbors:=∅;
            send(Q) to N(x);
            counter:=0;
            become ACTIVE;
        end

IDLE
        Receiving(Q)
        begin
            root:= false;
            parent:= sender;
            Tree-neighbors:={sender};
            send(Yes) to {sender};
            counter:=1;
            if counter=|N(x)| then
               become DONE
            else
               send(Q) to N(x) − {sender};
               become ACTIVE;
            endif
        end

ACTIVE
        Receiving(Q)
        begin
            send(No) to {sender};
        end

        Receiving(Yes)
        begin
            Tree-neighbors:=Tree-neighbors ∪{sender};
            counter:=counter+1;
            if counter=|N(x)| then become DONE; endif
        end

        Receiving(No)
        begin
            counter:=counter+1;
            if counter=|N(x)| then become DONE; endif
        end
```

Figure 9: *Shout protocol.*

Notice that:

- The **initiator** becomes the **root** of the ST;
- The **counter** is used to be sure that each node receives the answer from all the neighbors;
- For an IDLE node, if it is a **leaf** (condition counter = $N(x)$), then the only neighbor is the sender, so in this case he does not have to send any message;

- The ACTIVE state represents the case in which a node has **already** received a Q and replied with a *Yes*.

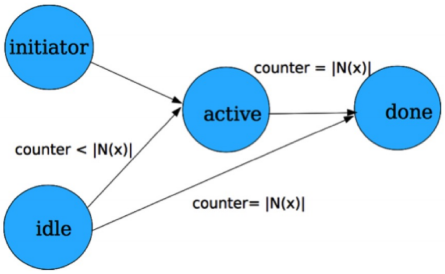


Figure 10: *States of the Shout protocol*

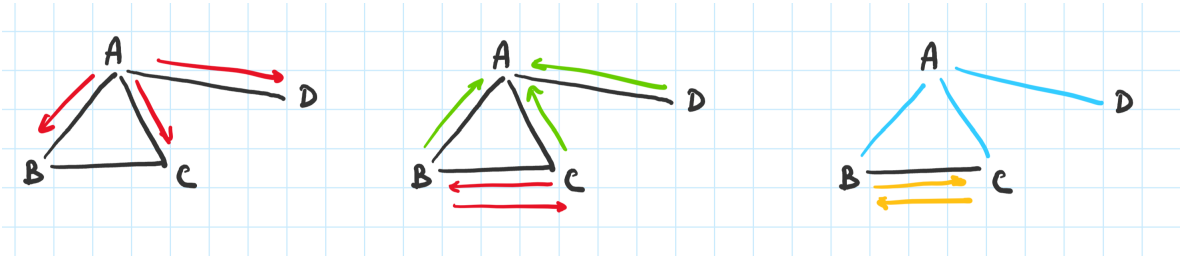


Figure 11: *Example of execution of the Shout protocol.*

Before we discuss the correctness and the efficiency of the protocol, consider how it is structured and operates. First of all observe that, in *Shout* the question Q is **broadcasted** through the network (using flooding). Further observe that, when an entity receives Q, it always sends a **reply** (either *Yes* or *No*). Summarizing, the structure of this protocol is a **flood** where every information message is acknowledged. This type of structure will be called *Flooding + Reply*.

Correctness The *Shout* protocol builds a Spanning Tree with tree-neighbors. Notice that the execution of protocol *Shout* ends with **local termination**: each entity knows when its own execution is over; this occurs when it enters status DONE. Notice however that **no entity**, including the initiator, **is aware of global termination** (i.e., every entity has locally terminated).

Costs The **message costs** of *Flooding + Reply*, and thus of *Shout*, are simple to analyze. As mentioned before, *Flooding + Reply* consists of an execution of *Flooding*(Q) with the addition of a reply (either *Yes* or *No*) for every Q. In other words

$$\mathbf{M}[\text{Flooding+Reply}] = 2\mathbf{M}[\text{Flooding}] = 2(2m - n + 1)$$

The **time costs** of *Flood + Reply*, and thus of *Shout*, are also simple to determine; in fact:

$$\mathbf{T}[\text{Flooding+Reply}] = T[\text{Flooding}] + 1$$

Thus

$$\mathbf{M}[\text{Shout}] = 4m - 2n + 2$$

$$\mathbf{T}[\text{Shout}] = r(s^*) + 1 \leq d + 1$$

Proof. As we can see from the image below, there exist some situations which can happen, and others that cannot. In particular:

- The **total number** of Q messages is given by the number of Q and *Yes* and the number of Q and Q.
  - The number of Q and *Yes* is equal to $n-1$, as the total number of links in the ST;
  - The number of Q and Q is $2(m-n+1)$, which are exactly the links that are not in the ST $(m-(n-1))$;
- The **total number** of *No* is given by the number of *No* and *No*, which is equal to the number of Q and Q, so $2(m-n+1)$;
- The **total number** of *Yes* is exactly $n-1$.

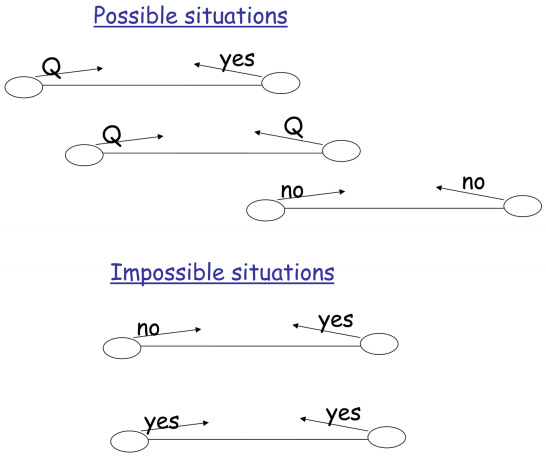


Figure 12: *Possible and impossible situations*

Finally, the **total number of messages** is given by:

$$\mathbf{M}[\text{Shout}] = Q + NO + YES = (n-1) + 2(m-n+1) + 2(m-n+1) + (n-1) = 4m - 2n + 2$$

Thus, $\Omega(m)$ represents a **lower bound** also in this case.

2.2.2 Shout+

Let us examine protocol *Shout* to see if it can be **improved**, thereby, helping us to save some messages. *Do we have to send No messages?*
When **constructing the spanning tree**, an entity needs to know who its tree-neighbors are; by construction, they are the ones that reply *Yes* and, except for the initiator, also the ones that first asked the question. Thus, for this determination, the *No* messages are **not needed**.
On the contrary hand, the *No* messages are **used** by the protocol to **terminate in finite time**. Consider an entity x that just sent Q to neighbor y; it is now waiting for a reply. If the reply is *Yes*, it knows y is in the tree; if the reply is *No*, it knows y is not. Should we remove the sending of *No*? How can x determine that y would have sent *No*?
More clearly: Suppose x has been waiting for a reply from y for a (very) long time; it does not know if y has sent *Yes* and the delays are very long, or y would have sent *No* and thus will send nothing. Because the algorithm must terminate, x cannot wait forever and has to make a decision. How can x decide?
The question is relevant because communication delays are finite but unpredictable. Fortunately, there is a simple answer to the question that can be derived by examining how protocol *Shout* operates.

Focus on a node x that just sent Q to its neighbor y. Why would y reply *No*? It would do so only if it had already said *Yes* to somebody else; if that happened, y sent Q at the same time to all its other neighbors, including x. Summarizing, if y replies *No* to x, it must have already sent Q to x. We can clearly use this fact to our advantage: after x sent Q to y, if it receives *Yes* it knows that y is its neighbor in the tree; if it receives Q, it can deduce that y will definitely reply *No* to x's question. All of this can be deduced by x without having received the *No*.
In other words: a **message** Q that arrives at a node **waiting for a reply** can act as an **implicit negative acknowledgment**; therefore, we can **avoid** sending *No* messages.

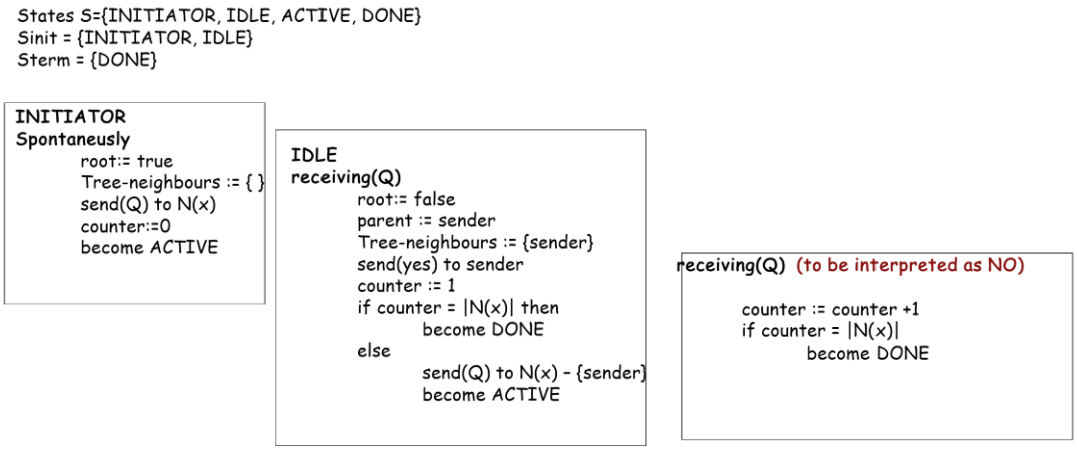


Figure 13: *Shout+ protocol.*

In this case, on each link there will be exactly 2 messages: either Q and Q or Q and *Yes*. Thus, the total number of messages is:

$$\mathbf{M}[\text{Shout+}] = 2m$$

which is much better than $4m - 2n + 2$.

2.2.3 SPT construction with multiple initiators

We have started examining the spanning-tree construction assuming that there is a **unique initiator**. This is unfortunately a very strong (and "unnatural") assumption to make, as well as difficult and expensive to guarantee.
What happens to the single-initiator protocols *Shout* if there is **more than one initiator**?
Let us examine first protocol *Shout*. Consider the very simple case of three entities, x, y, and z, connected to each other. Let both x and y be initiators and start the protocol, and let the Q message from x to z arrive there before the one sent by y.
In this case, neither the link (x, y) nor the link (y, z) will be included in the tree; hence, the algorithm creates not a spanning tree but a spanning forest, which is not connected. Thus, another protocol has to be devised, or an election is needed to have a unique initiator.

2.3 Computations on trees

In this section, we consider computations in tree networks under the standard restrictions R (bidirectional links, ordered messages and full reliability) plus clearly the common knowledge that the network is **tree**.
Note that the knowledge of being in a tree implies that each **entity** can determine whether it is a **leaf** (i.e., it has only one neighbor) or an **internal node** (i.e., it has more than one neighbor).

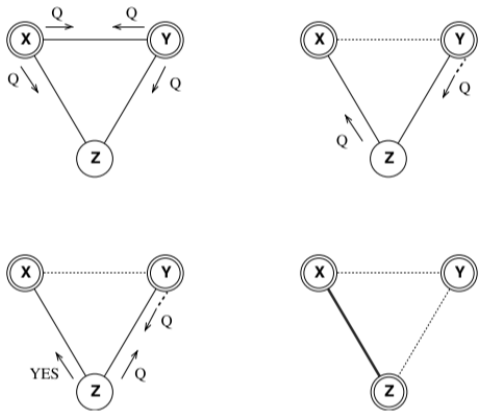


Figure 14: *Shout protocol with multiple initiators.*

We need to introduce some basic **concepts** and terminology about trees. In a tree T, the removal of a link (x, y) will disconnect T into two trees, one containing x (but not y), the other containing y (but not x); we shall denote them by $T[x - y]$ and $T[y - x]$, respectively. Let $d[x, y] = \max d(x, z) : z \in T[y - x]$ be the longest distance between x and the nodes in $T[y - x]$. Recall that the longest distance between any two nodes is called diameter, and it is denoted by d. If $d[x, y] = d$, the path between x and y is said to be diametral.

2.3.1 Saturation: a basic technique

The technique, which we shall call *Full Saturation*, is very simple and can be autonomously and independently started by any number of initiators. It is composed of three stages:

1. The **activation** stage, started by the initiators, in which all nodes are activated;
2. The **saturation** stage, started by the leaf nodes, in which a unique couple of neighboring nodes is selected;
3. The **resolution** stage, started by the selected pair.

The activation stage is just a **wake-up**: each initiator sends an activation (i.e., wake-up) message to all its neighbors and becomes active; any non initiator, upon receiving the activation message from a neighbor, sends it to all its other neighbors and becomes active; active nodes ignore all received activation messages. Within finite time, all nodes become active, including the leaves. The **leaves** will **start** the **second stage**.
Each active **leaf** starts the saturation stage by sending a **message** (call it M) to its only **neighbor**, referred now as its "parent" and becomes processing. (Note: M messages will start arriving within finite time to the internal nodes.) An **internal node** waits until it has received an M message from all its neighbors but one, sends a M message to that neighbor that will now be considered its "parent," and becomes **processing**. If a processing node receives a message from its parent, it becomes **saturated**.
The **resolution** stage is started by the **saturated nodes**; the nature of this stage depends on the application. Commonly, this stage is used as a notification for all entities (e.g., to achieve local termination). Since the nature of the final stage will depend on the application, we will only describe the set of rules implementing the first two stages of *Full Saturation*.

Lemma. *Exactly* ***two processing nodes will become saturated****; furthermore, these two nodes are neighbors and are each other's parent.*

- Status: $\mathcal{S}$ = {AVAILABLE, ACTIVE, PROCESSING, SATURATED};
  $\mathcal{S}_{INIT}$ = {AVAILABLE};
- Restrictions: **R** ∪ T.

```
AVAILABLE
        Spontaneously
        begin
            send(Activate) to N(x);
            Initialize;
            Neighbors:= N(x);
            if|Neighbors|=1 then
              Prepare_Message;
              parent ⇐ Neighbors;
              send(M) to parent;
              become PROCESSING;
            else become ACTIVE;
            endif
        end

        Receiving(Activate)
        begin
            send(Activate) to N(x) − {sender};
            Initialize;
            Neighbors:= N(x);
            if|Neighbors|=1 then
              Prepare_Message;
              parent ⇐ Neighbors;
              send(M) to parent;
              become PROCESSING;
            else become ACTIVE;
            endif
        end

ACTIVE
        Receiving(M)
        begin
            Process_Message;
            Neighbors:= Neighbors−{sender};
            if|Neighbors|=1 then
              Prepare_Message;
              parent ⇐ Neighbors;
              send(M) to parent;
              become PROCESSING;
            endif
        end

PROCESSING
        Receiving(M)
        begin
            Process_Message;
            Resolve;
        end

Procedure Initialize
begin
    nil;
end

Procedure Prepare_Message
begin
    M:=("Saturation");
end

Procedure Process_Message
begin
    nil;
end

Procedure Resolve
begin
    become SATURATED;
    Start Resolution stage;
end
```

Figure 15: *Full saturation protocol and procedures.*

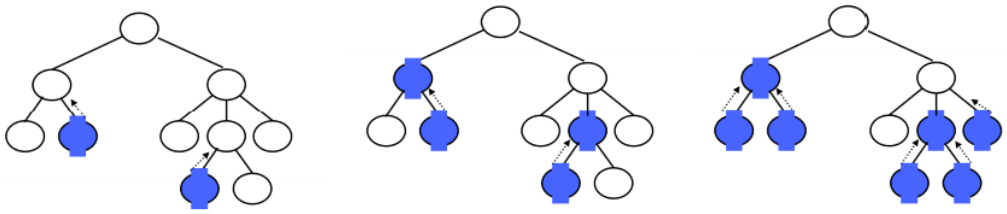


Figure 16: *Example - part 1.*

Proof. From the algorithm, it follows that an entity sends a message M only to its parent and becomes saturated only upon receiving an M message from its parent. Choose an arbitrary node x, and traverse the "up" edge of x (i.e., the edge along which the M message was sent from x to its parent). By moving along "up" edges, we must meet a saturated node s_1 since there are no cycles in the graph. This node has become saturated when receiving an M message from its parent s_2. Since s_2 has sent an M message to s_1, this implies that s_2 must have been processing and must

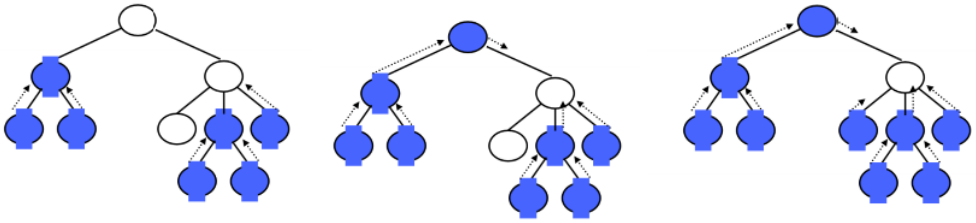


Figure 17: *Example - part 2.*

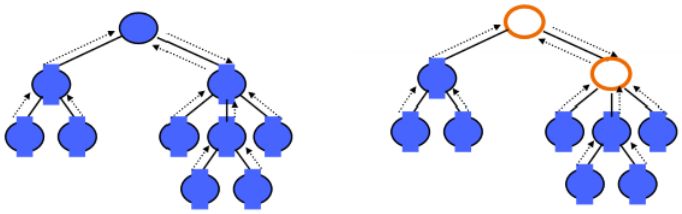


Figure 18: *Example - part 3.*

have considered s_1 its parent; thus, when the M message from s_1 will arrive at s_2, s_2 will become saturated also. Thus, there exist at least two nodes that become saturated; furthermore, these two nodes are each other's parent. Assume that there are more than two saturated nodes; then there exist two saturated nodes, x and y, such that $d(x, y) \geq 2$. Consider a node z on the path from x to y; z could not send an M message toward both x and y; therefore, one of the nodes cannot be saturated. Therefore, the lemma holds.
It is important to notice that it depends on the communication delays which entities will become saturated and it is therefore totally **unpredictable**. Subsequent executions with the same initiators might generate different results. In fact, any pair of neighbors could become saturated. The only guarantee is that a pair of neighbors will be selected; since a pair of neighbors uniquely identifies an edge, the one connecting them; this result is also called edge election.

Message complexity To determine the number of message exchanges, observe that the **activation** stage is a wake-up in a tree and hence it will use $2(n-1)$ messages if there are n initiators. During the **saturation** stage, exactly one message is transmitted on each edge, except the edge connecting the two saturated nodes on which two M messages are transmitted, for a total of $n - 1 + 1 = n$ messages. Finally, in the **notification** stage, $n - 2$ messages are sent. Thus,

$$\mathbf{M}[\text{Full Saturation}] = 2n - 2 + n + n = 4n - 4 = O(n)$$

In general, for $i < n$ initiators, we have the number of messages in the wake up phase is:

$$\sum_{x \in S} |N(x)| + \sum_{x \notin S} (|N(x)| - 1) = \sum_{x} |N(x)| - \sum_{x \notin S} 1 = 2(n-1) - (n-i) = n + i - 2$$

There are other n messages in the saturation phase, and $n-2$ in the notification phase, thus the total number of messages is:

$$\mathbf{M}[\text{Full Saturation}] = n + i - 2 + n + n - 2 = 3n + i - 4$$

2.3.2 Minimum finding

Let us see how the saturation technique can be used to compute the **smallest** among a set of values distributed among the nodes of the network. Every entity x has an input value $v(x)$ and is initially in the **same status**; the task is to determine the **minimum** among those input values. That is, in the end, each entity must know whether or not its value is the smallest and enter the appropriate status, minimum or large, respectively.
It is important to notice that these values are **not necessarily distinct**. So, more than one entity can have the minimum value; all of them must become minimum. This problem is called Minimum Finding (*MinFind*).
Full Saturation allows to achieve the same goals without a root or any additional information. This is achieved simply by **including** in the M message the **smallest value known to the sender**. Namely, in the saturation stage the leaves will send their value with the M message, and each internal node sends the smallest among its own value and all the received ones. In other words, *MinF-Tree* is just protocol *Full Saturation* where the procedures *Initialize*, *Prepare Message*, and *Process Message* are as shown below and where the **resolution** stage is just a **notification** started by the two saturated nodes, of the minimum value they have computed. This is obtained by simply modifying procedure *Resolve* accordingly and adding the rule for handling the reception of the notification.

```
PROCESSING
        Receiving (Notification)
        begin
            send (Notification) to N(x)−parent;
            if   v(x) =Received_Value then
                 become MINIMUM;
            else
                 become LARGE;
            endif
        end

Procedure Initialize
begin
    min:=v(x);
end

Procedure Prepare_Message
begin
    M:=("Saturation", min);
end

Procedure Process_Message
begin
    min:= MIN{min, Received_Value};
end

Procedure Resolve
begin
    Notification:= ("Resolution", min);
    send (Notification) to N(x)−parent;
    if   v(x) =min then
         become MINIMUM;
    else
         become LARGE;
    endif
end
```

Figure 19: *New rule and procedures used for Minimum Finding.*

Example. *An example of the Minimum Finding protocol follows.*

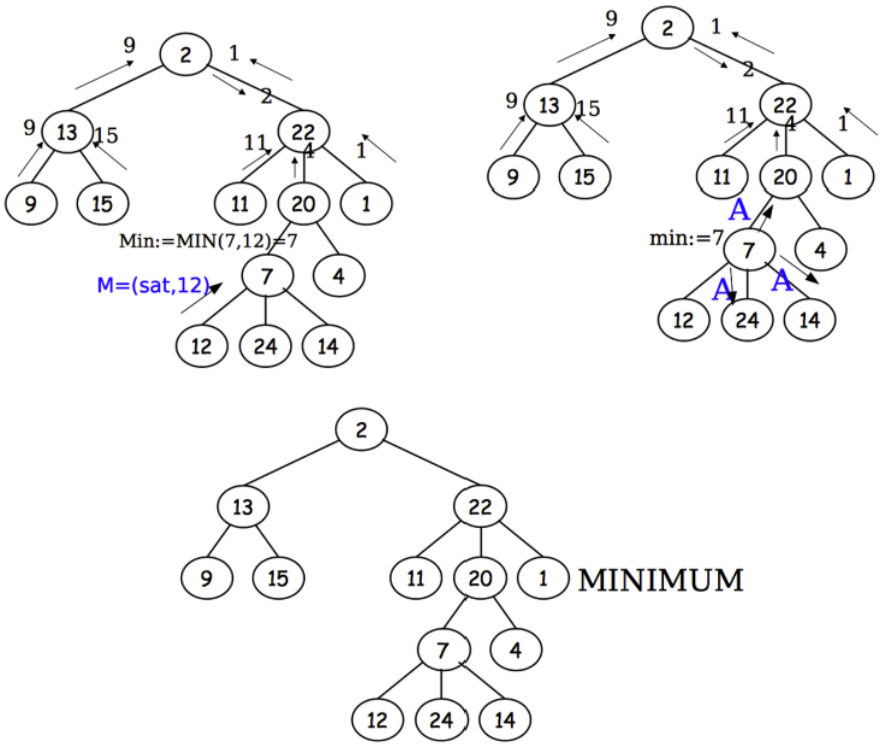


Figure 20: *Example of execution.*

The **correctness** follows from the fact that both saturated nodes know the minimum value.

Message complexity The number of message transmission for the minimum-finding algorithm *MinF-Tree* will be exactly the same as the one experienced by *Full Saturation* plus the ones performed during the notification. Since a notification message is sent on every link except the one connecting the two saturated nodes, there will be exactly $n-2$ such messages. Hence

$$\mathbf{M}[\text{MinF-Tree}] = \mathbf{M}[\text{Full Saturation}] = 4n - 4 = O(n)$$

Time complexity The time costs will be the one experienced by *Full Saturation* plus the ones required by the notification. Let *Sat* denote the set of the two saturated nodes; then

$$\mathbf{T}[\text{MinF-Tree}] = \mathbf{T}[\text{Full Saturation}] + \max d(s,x) : s \in Sat, x \in V$$

2.3.3 Ranking problem

Given a node x whose value is $v(x)$, the **rank** is defined as:

$$rank(x) = 1 + |y \in V : v(y) < v(x)|$$

, i.e. the number of nodes whose values are smaller than the one of x.
Given an arbitrary network, the operations are the following:

1. Find a **spanning tree**;
2. Use **Saturation** and **Minimum Finding** to find a starting node;
3. Do **Ranking** (centralized or decentralized).

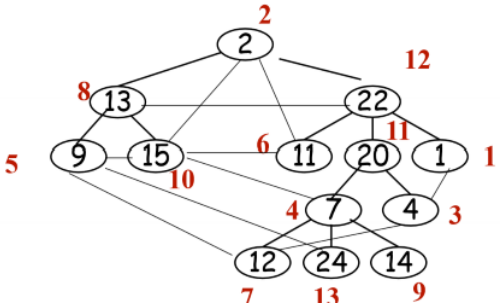


Figure 21: *Ranking problem.*

Centralized ranking

1. We **elect a leader** (one of the saturated nodes): the leader knows the **minimum** (he might not be the minimum), it sends in that direction a ranking message;
2. Every node knows the minimum in its **subtrees**, it can then forward the ranking message (ranking,minimum) in the right direction;
3. When the node to be ranked receives the message it **sends back** a **notification&update message** (new-minumum) that will travel up to the leader.

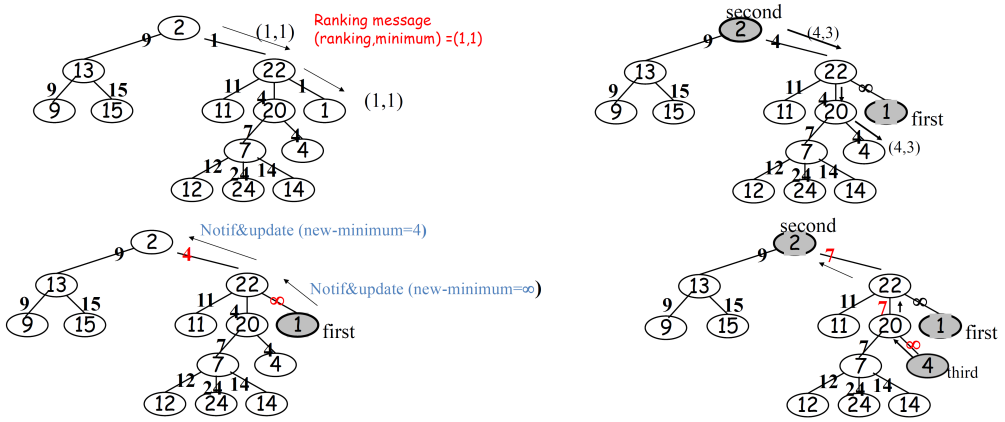


Figure 22: *Example of execution of the Ranking protocol.*

The **worst case** for this protocol is given by the following situation.

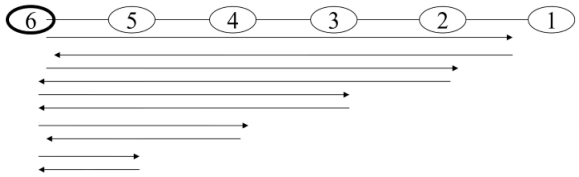


Figure 23: *Worst case scenario for centralized ranking.*

In this case:

$$\mathbf{M}[\text{Centralized Ranking}] = 2(n-1) + 2(n-2) + .. = \frac{2(n-1)n}{2} = (n-1)n = O(n^2)$$

Decentralized ranking In this case, the starter node sends a ranking message of the form (first, second,rank) in the direction of first:

- **First**: smallest value;
- **Second**: second smallest known so far (this is a guess on the value that has to be ranked after first).

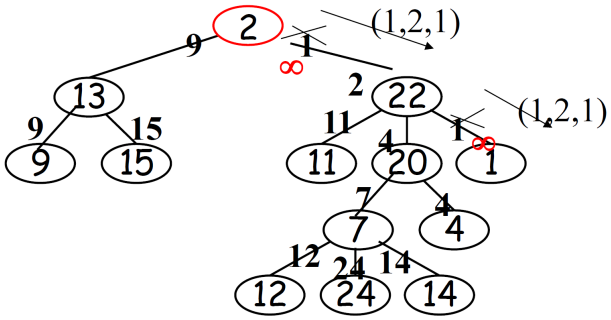
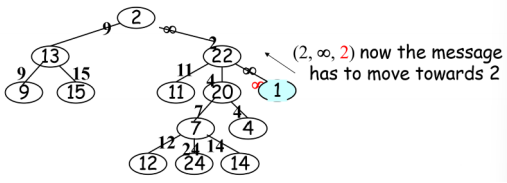


Figure 24: *Decentralized ranking.*

In this case, the **value** on a link indicates the **smallest value in the corresponding subtree**. If no value is indicated (or the value is ∞) it means that the smallest value in the corresponding subtree is **unknown** (for the moment).

Example. *In this case, the ranked node attempts to send a ranking message to the next node to be ranked. Since the second node might now be unknown, in this case the value ∞ is used. The*



second variable of the rank message is updated during its travel, and the minimum values on the links of the tree are also updated.

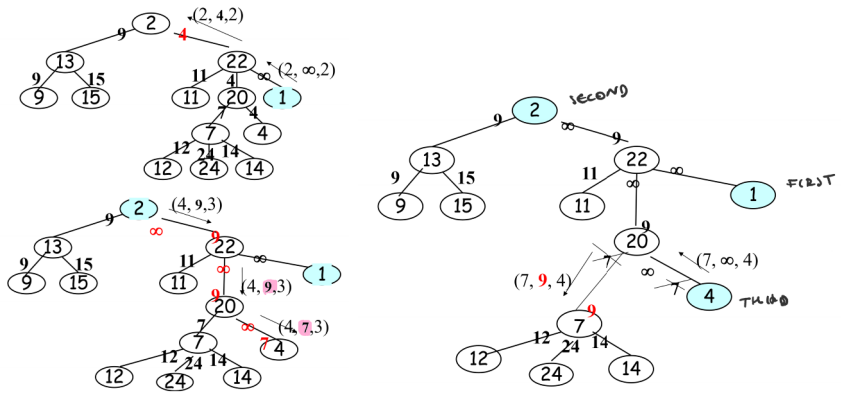


Figure 25: *Example.*

In this case, the **worst case** is the one in which we have to go back and forth.

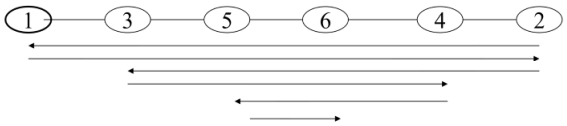


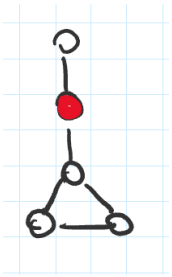
Figure 26: *Worst case scenario for decentralized ranking.*

The number of messages is:

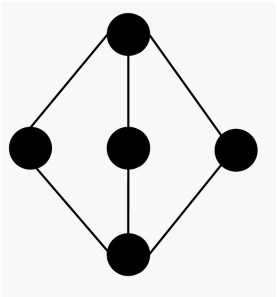
$$\mathbf{M}[\text{Decentralized Ranking}] = 2(n-1) + (n-2) + (n-3) = \frac{n(n-1)}{2} + (n-1) = \frac{n-1}{n/2-1}$$

2.4 Exercises

1. Draw a hypercube of dimension 3. What is a simple and efficient algorithm to do broadcasting in the hypercube? Describe it (with code, pseudocode, words, as long a it is complete, precise and clear). Provide an execution example on the hypercube of point 1;
2. Run the Shout algorithm on this graph;

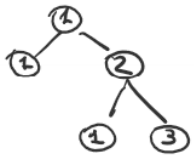


3. Show an execution of the Shout algorithm (for the Spanning Tree construction) on this graph. Solution on slide 6-10 of L18;

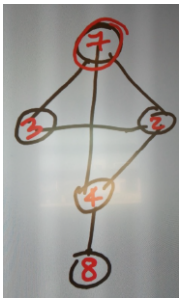


4. Run the Shout+ algorithm on the graph on the left, and the Saturation algorithm on the graph on the right;

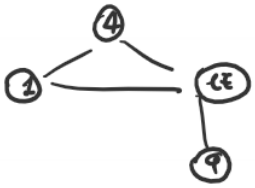
5. Given an arbitrary graph of n nodes, where each node contains an integer value: describe a complete and precise distributed algorithm (in code, or pseudocode or words), that multiplies all the n values stored in the nodes, show how the algorithm works on a small example and state the complexity of the algorithm;
6. Show an execution of the Saturation algorithm on this graph. Solution on slide 14-20 of L18;



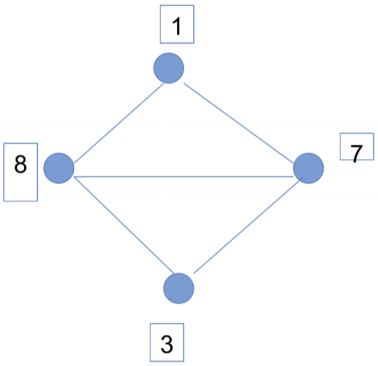
7. Run the Minimum Finding algorithm on this graph. Solution on slide 1 of L17;
8. Rank the nodes of the following graph using the centralized and decentralized ranking algorithm;



9. Rank the nodes of the following graph using the centralized and decentralized ranking algorithm. Solution on slide 2 of L17;



10. Rank the nodes of the following graph using the decentralized ranking algorithm



3 Election

3.1 Introduction

In a distributed environment, most applications often require a single entity to act temporarily as a central controller to coordinate the execution of a particular task by the entities. In some cases, the need for a **single coordinator** arises from the desire to simplify the design of the solution protocol for a rather complex problem; in other cases, the presence of a single coordinator is required by the nature of the problem itself.
The problem of **choosing** such a **coordinator** from a population of autonomous symmetric entities is known as Leader Election (**Elect**). Formally, the task consists in moving the system from an initial configuration where all entities are in the **same state** (usually called *available*) into a final configuration where all entities are in the same state (traditionally called *follower*), **except one**, which is in a different state (traditionally called *leader*). There is no restriction on the number of entities that can start the computation, nor on which entity should become leader.
As election provides a mechanism for **breaking the symmetry** among the entities in a distributed environment, it is at the base of most control and coordination processes (e.g., mutual exclusion, synchronization, concurrency control, etc.) employed in distributed systems, and it is closely related to other basic computations (e.g., minimum finding, spanning-tree construction, traversal).

3.1.1 Impossibility result

We will start considering this problem under the standard restrictions R: Bidirectional Links, Connectivity, and Total Reliability. There is unfortunately a very strong impossibility result about election.

Theorem. *Problem Elect is* ***deterministically unsolvable*** *under R.*

In other words, there is **no deterministic protocol** that will always correctly terminate within finite time if the only restrictions are those in R.
To see why this is the case, consider a simple system composed of two entities, x and y, both initially available and with no different initial values; in other words, they are initially in identical states. If a solution protocol P exists, it must work under any conditions of message delays. Consider a synchronous schedule (i.e., an execution where communication delays are unitary) and let the two entities start the execution of P simultaneously. As they are in identical states, they will execute the same rule, obtain the same result, and compose and send (if any) the same message; thus, they will still be in identical states. If one of them receives a message, the other will receive the same message at the same time and they will perform the same computation, and so on. Their state will always be the same; hence if one becomes leader, so will the other. But this is against the requirement that there should be only one leader; in other words, P is not a solution protocol.
Another technique that can be used to build a ST of a graph is based on the DFS (Depth First Search) algorithm: this technique is easy, but has the worst time complexity.

3.1.2 Solution strategies

To each node x is associated a value $v(x)$, thus a simple algorithm could be:

1. Execute the **saturation** technique;
2. **Choose** the saturated node holding the **minimum value**.

3.2 Election in rings

We will now consider a network topology that plays a very important role in distributed computing: the **ring**, sometimes called **loop network**.
A ring consists of a single cycle of length n. In a ring, each entity has exactly two neighbors, (whose associated ports are) traditionally called left and right.

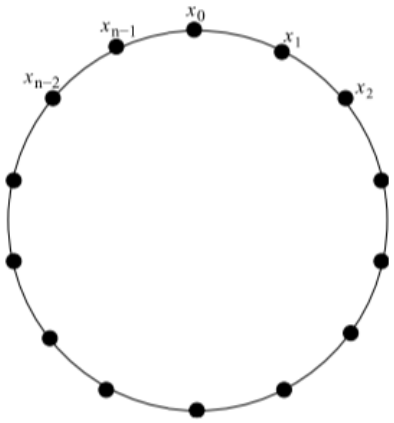


Figure 27: *A ring network.*

After trees, rings are the networks with the **sparsest** topology: $m = n$; however, unlike trees, rings have a complete **structural symmetry** (i.e., all nodes look the same).
We will denote the ring by $R = (x_0, x_1, .., x_{n-1})$. Let us consider the problem of electing a leader in a ring R, under the standard set of restrictions for election, **IR** = Bidirectional Links, Connectivity, Total Reliability, Initial Distinct Values, as well as the knowledge that the network is a ring (**Ring**). Denote by $id(x)$ the unique value associated to x.
Because of its structure, in a ring we will use almost exclusively the approach of minimum finding as a tool for leader election. In fact we will consider both the *Elect Minimum* (find smallest value and elect as a leader the node with such a value) and the *Elect Minimum Initiator* (find smallest value among the initiators and elect as a leader the node with such a value) approaches. Clearly the first solves both Min and Elect, while the latter solves only Elect.
Every protocol that elects a leader in a ring can be made to find the minimum value (if it has not already been determined) with an additional n message and time. Furthermore, in the worst case, the two approaches coincide: All entities might be initiators.

3.2.1 All the Way

The first solution we will use is rather straightforward: When an entity starts, it will **choose** one of its two neighbors and **send** to it an "Election" **message** containing its *id*; an entity receiving the *id* of somebody else will send its *id* (if it has not already done so) and **forward** the received message along the ring (i.e., send it to its other neighbor) keeping track of the smallest *id* seen so far (including its own).
This process can be visualized as follows: Each entity originates a message (containing its *id*), and this **message travels "all the way" along the ring** (forwarded by the other entities).
Each entity will eventually see the *id* of everybody else *id* (finite communication delays and total reliability ensure that) including the minimum value; it will, thus, be able to determine whether or not it is the (unique) **minimum** and, thus, the leader.
When will this happen? In other words, *When will an entity terminate its execution?*

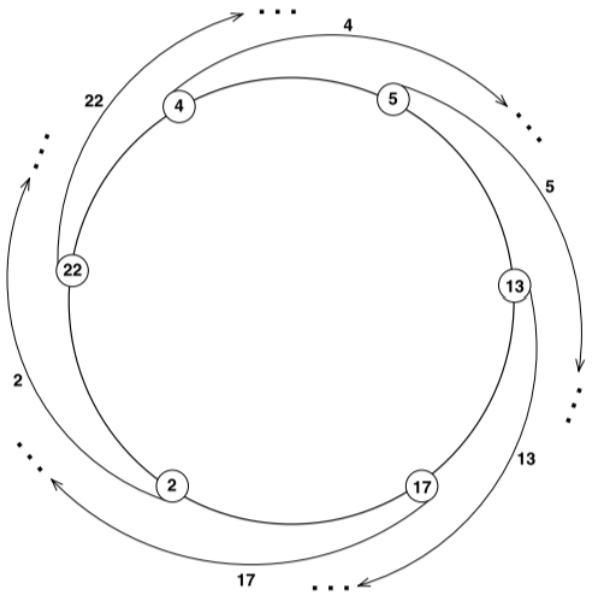


Figure 28: *All the Way. Every id travels along the ring.*

Entities only forward messages carrying values other than their own: Once the message with $id(x)$ arrives at x, it is no longer forwarded. Thus, each **value** will **travel** "All the Way" along the ring **only once**. So, the communication activities will **eventually terminate**. But how does an entity know that the communication activities have terminated, that no more messages will be arriving, and, thus, the smallest value seen so far is really the minimum *id*?
Consider a "reasonable" but unfortunately incorrect answer:
An entity knows that it has seen all values once it receives its value back.
The "reason" is that the message with its own *id* has to travel longer along the ring to reach x than those originated by other entities; thus, these other messages will be received first. In other words, **reception** of its own message can be **used** to **detect termination**.
This reasoning is **incorrect** because it uses the (hidden) additional assumption that the system has first in first out (**FIFO**) communication channels, that is, the messages are delivered in the order in which they arrive. This restriction, called **Message Ordering**, is not a part of election's standard set; few systems actually have it built in, and the costs of offering it can be formidable. So, whatever the answer, it must not assume FIFO channels. With this proviso, a "reasonable" but unfortunately still incorrect answer is the following:
An entity counts how many different values it receives; when the counter is equal to n, it knows it can terminate.
The problem is that this answer assumes that the entity knows n, but a priori **knowledge** of the **ring size** is **not** a **part** of the **standard restrictions** for election. So it cannot be used.
It is indeed strange that the termination should be difficult for such a simple protocol in such a clear setting. Fortunately, the last answer, although incorrect, provides us with the way out. In fact, although n is not known a priori, it can be **computed**. This is easily accomplished by having a **counter** in the Election message, initialized to 1 and **incremented by each entity forwarding it**; when an entity receives its id back, the value of the counter will be n.
Summarizing, we will use a **counter** at each **entity**, to keep track of how many different *ids* are received and a **counter** in each **message**, so that each entity can determine n.

Protocol

PROTOCOL All the Way.

- States: $\mathcal{S}$ = {ASLEEP, AWAKE, FOLLOWER, LEADER};
  $\mathcal{S}_{\text{INIT}}$ = {ASLEEP};
  $\mathcal{S}_{\text{TERM}}$ = {FOLLOWER, LEADER}.
- Restrictions: **IR** $\cup$ *Ring*.

```
ASLEEP
      Spontaneously
      begin
           INITIALIZE;
           become AWAKE;
      end

      Receiving("Election", value*, counter*)
      begin
           INITIALIZE;
           send("Election", value*, counter*+1) to other;
           min:= Min{ min, value};
           count:= count+1;
           become AWAKE;
      end


AWAKE
      Receiving("Election", value*, counter*)
      begin
           if value ≠ id(x) then
             send("Election", value*, counter*+1) to other;
             min:= MIN{min,value*};
             count:= count+1;
             if known then CHECK endif;
           else
             ringsize:= counter*;
             known:= true;
             CHECK;
           endif
      end

      Procedure INITIALIZE
      begin
           count:= 0;
           size:= 1;
           known:= false;
           send("Election", id(x), size) to right;
           min:= id(x);
      end

      Procedure CHECK
      begin
           if count = ringsize then
             if min = id(x) then
                become LEADER;
             else
                become FOLLOWER;
             endif
           endif
      end
```

Figure 29: *Protocol and procedures of All the Way.*

Complexity The message originated by each entity will travel along the ring exactly **once**. Thus, there will be exactly n^2 messages in total, each carrying a counter and a value, for a total of $n^2 \log(id + n)$ bits. The time costs will be at most $2n$.
Summarizing,

$$\mathbf{M}[\text{AllTheWay}] = n^2$$

$$\mathbf{T}[\text{AllTheWay}] \leq 2n - 1$$

In particular, the **bound** of the time is given by the fact that we have just **one initiator** that wakes up one node at a time, thus $n - 1$ nodes to wake up, and n to get an answer, giving $\leq 2n - 1$.

The solution protocol we have just designed is **very expensive** in terms of **communication costs** (in a network with 100 nodes it would cause 10, 000 message transmissions). Notice that *All the Way* (in its original or modified version) can be used also in **unidirectional rings** with the same costs. In other words, it does not require the Bidirectional Links restriction. We will return to this point later.

3.2.2 As Far As It Can

To design an improved protocol, let us determine the drawback of the one we already have: *All the Way*. In this protocol, each message travels all along the ring.
Consider the situation of a message containing a large *id*, say 22, arriving at an entity x with a smaller *id*, say 4. In the existing protocol, x will forward this message, even though x knows that 22 is not the smallest value.
But our overall strategy is to determine the smallest *id* among all entities; if an entity determines that an *id* is not the minimum, there is **no need** whatsoever for the message containing such an *id* to **continue traveling** along the ring.

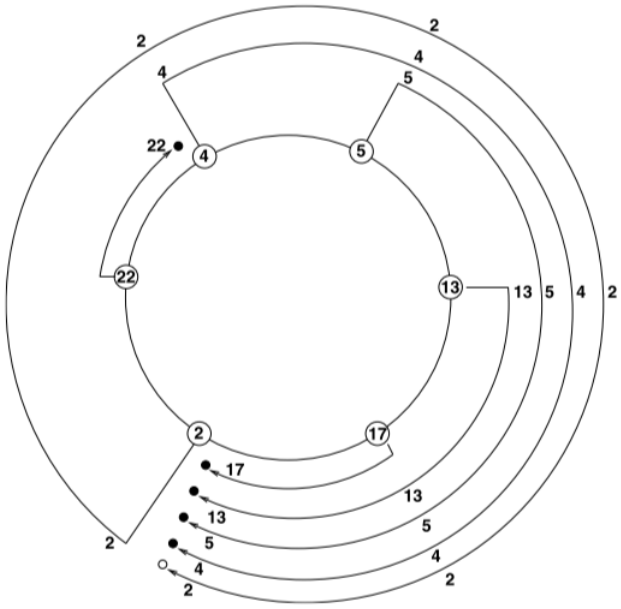
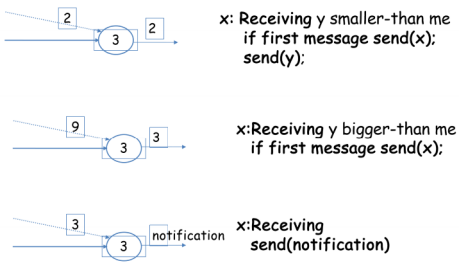


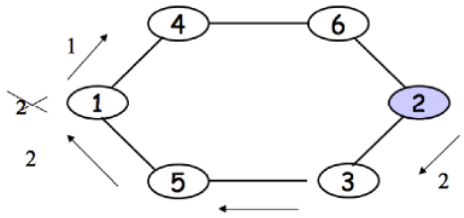
Figure 30: *As Far As It Can. Message with a larger id does not need to be forwarded.*

We will thus **modify** the original protocol *All the Way* so that an **entity** will only **forward** Election messages carrying an *id* **smaller** than the **smallest seen so far** by that entity. In other words, an entity will become an insurmountable obstacle for all messages with a larger *id* "terminating" them.
Let us examine what happens with this simple modification. Each entity will originate a message (containing its *id*) that travels along the ring **"as far as it can"**: until it returns to its originator or arrives at a node with a smaller *id*.
When will an entity terminate its execution? The message with the smallest *id* will always be forwarded by the other entities; thus, it will travel all along the ring returning to its originator. The message containing another *id* will instead be unable to return to its originator because it will find an entity with a smaller *id* (and thus be terminated) along the way. In other words, only the **message** with the **smallest** *id* will **return** to its **originator**. This fact provides us with a **termination detection mechanism**.



If an entity receives a message with its own *id*, it knows that its *id* is the **minimum**, that is, it is the **leader**; the other entities have all seen that message pass by (they forwarded it) but they still do not know that there will be no smaller ids to come by. Thus, to ensure their termination, the newly elected leader must **notify** them by sending an additional message along the ring.

Example. *This is how value 2 travels.*



Correctness and termination The leader knows it is the leader when it receives its message back. *When do the other nodes know?* **Notification** is necessary (since I do not receive my message back if I am not the leader, I am also not using counters).
Correctness: the message with smaller identity starts and comes back to the sender which can then send the notification (bidirectional version).

Space Complexity This protocol will definitely have **fewer messages** than the previous one. The **exact number** depends on several factors. Consider the cost caused by the Election message originated by x. This message will travel along the ring until it finds a smaller *id* (or complete the tour). Thus, the **cost** of its travel **depends** on how the *ids* are **allocated** on the ring. Also notice that what matters is whether an *id* is smaller or not than another and not their actual value.
In other words, what is important is the **rank** of the *ids* and how those are **situated** on the ring. Denote by $\#i$ the *id* whose rank is i.
Worst Case Let us first consider the **worst possible case**. Id #1 will always travel all along the ring costing n messages. Id #2 will be stopped only by id #1; so its cost in the worst case is $n-1$, achievable if id #2 is located immediately after id #1 in the direction it travels.
In general, id $\#(i+1)$ will be **stopped** by any of those with **smaller rank**, and, thus, it will cost at most $n-i$ messages; this will happen if all those **entities** are **next to each other**, and id $\#(i+1)$ is located **immediately after** them in the direction it will travel. In fact, all the worst cases for each of the *ids* are simultaneously achieved when the *ids* are arranged in an **(circular) order according to their rank** and all **messages** are **sent** in the **"increasing" direction**.

PROTOCOL AsFar.

- States: $\mathcal{S}$ = {ASLEEP, AWAKE, FOLLOWER, LEADER};
  $\mathcal{S}_{\text{INIT}}$ = {ASLEEP};
  $\mathcal{S}_{\text{TERM}}$ = {FOLLOWER, LEADER}.
- Restrictions: **IR** $\cup$ *Ring*.

```
ASLEEP
      Spontaneously
      begin
          INITIALIZE;
          become AWAKE;
      end

      Receiving("Election", value)
      begin
          INITIALIZE;
          if value < min then
            send("Election", value) to other;
            min:= value;
          endif
          become AWAKE;
      end


AWAKE
     Receiving("Election", value)
     begin
         if value < min then
           send("Election", value) to other;
           min:= value;
         else
           if value min then NOTIFY endif;
         endif
     end

     Receiving(Notify)
         send(Notify) to other;
         become FOLLOWER;
     end
```

where the procedures *Initialize* and *Notify* are as follows:

```
Procedure INITIALIZE
begin
    send("Election", id(x)) to right;
    min:= id(x);
end

Procedure NOTIFY
begin
    send(Notify) to right;
    become LEADER;
end
```

Figure 31: *Protocol and procedures of AsFar.*

In this case, including also the n messages required for the final notification, the total cost will be:

$$\mathbf{M}[\text{AsFar}] = n + \sum_{i=1}^{n} i = \frac{n(n+3)}{2} = O(n^2)$$

That is, we will **cut** the **number of messages** at least to **half**. From a theoretical point of view, the improvement is not significant; from a practical point of view, this is already a reasonable

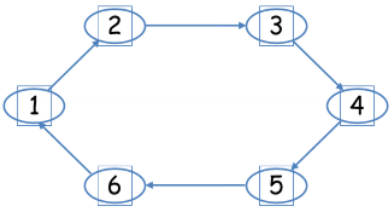


Figure 32: *Worst-case complexity (unidirectional version).*

achievement. However we have so far analyzed only the worst case. In general, the improvement will be much more significant. To see precisely how, we need to perform a more detailed analysis of the protocol's performance.
Notice that *AsFar* can be used in **unidirectional rings**. In other words, it does not require the Bidirectional Links restriction.We will return to this point later.
Average Case We will first consider the case when the ring is **oriented**, that is, "right" means the same to all entities. In this case, all messages will travel in only one direction, say clockwise. Because of the unique nature of the ring network, this case coincides with the **execution** of the **protocol** in a **unidirectional ring**. Thus, the results we will obtain will hold for those rings.
To determine the average case behavior, we consider **all possible arrangements** of the ranks $1, ..., n$ in the ring as **equally likely**. Given a set of size a, we denote by $C(a, b)$ the number of subsets of size b that can be formed from it.
Consider the id $\#i$ with rank i; it will travel clockwise exactly k steps if and only if the ids of its $k-1$ clockwise neighbors are larger than it (and thus will forward it), while the id of its k-th clockwise neighbor is smaller (and thus will terminate it).
There are $i-1$ ids smaller than id $\#i$ from which to choose those $k-1$ smaller clockwise neighbors, and there are $n-i$ ids larger than id $\#i$ from which to choose the k-th clockwise neighbor. In other words, the number of situations where id $\#i$ will travel clockwise exactly k **steps** is $C(i-1, k-1)C(n-i, 1)$, out of the total number of $C(n-1, k-1)C(n-k, 1)$ possible situations.
Thus, the **probability** $P(i, k)$ that id $\#i$ will travel clockwise exactly k steps is:

$$P(i,k) = \frac{C(i-1,k-1)C(n-i,1)}{C(n-1,k-1)C(n-k,1)}$$

The smallest id, #1, will travel the full length n of the ring. The id $\#i$, $i > 1$, will travel less; the **expected distance** will be:

$$E_i = \sum_{k=1}^{n-1} kP(i,k)$$

Therefore, the overall **expected number of message** transmissions is:

$$E_i = n + \sum_{i=1}^{n-1}\sum_{k=1}^{n-1} kP(i,k) = n + \sum_{k=1}^{n-1} \frac{n}{k+1} = nH_n$$

where $H_n = 1 + \frac{1}{2} + \frac{1}{3} + .. + \frac{1}{m}$ is the n-th Harmonic number.
More precisely, $H_n = \ln n + O(1) \approx 0.69 \log n$, thus:

Theorem. *In oriented and in unidirectional rings, protocol AsFar will cost* $nH_n \approx 0.69n \log n + O(n)$ ***messages*** *on an* ***average****.*

This is indeed great news: On an average, the message cost is an order of magnitude less than that in the worst case. For $n = 1024$, this means that on an average we have 7066 messages instead of 525, 824, which is a considerable difference.
Let us now consider what will happen on an average in the general case, when the ring is **unoriented**. As before, we consider all possible arrangements of the ranks $1, ..., n$ of the values in the ring as equally likely. The fact that the ring is not oriented means that when two entities send a message to their "right" neighbors, they might send it in different directions.

Theorem. *In unoriented rings, Protocol AsFar will cost* $\approx 0.49n \log n$ ***messages*** *on an* ***average****.*

Time Complexity The time costs are the same as the ones of *All the Way* plus an additional n for the notification phase.

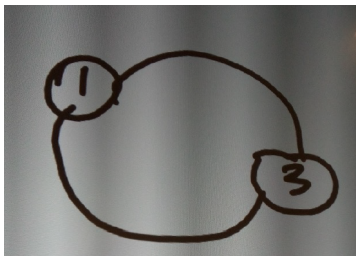
$$\mathbf{T}[\text{As Far}] \leq 2n - 1 + n = 3n - 1$$

This can, however, be halved by exploiting the fact that the links are **bidirectional** and by **broadcasting** the notification; this will require an extra message but halve the time.

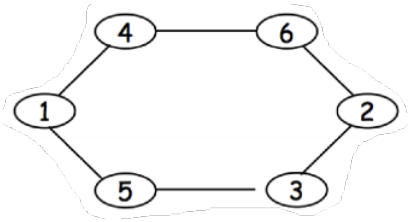
Summary The main **drawback** of protocol *AsFar* is that there still exists the possibility that a **very large number of messages** ($O(n^2)$) will be exchanged. As we have seen, on an average, the use of the protocol will cost only $O(n \log n)$ messages. There is, however, **no guarantee** that this will happen the next time the protocol will be used. To give such a guarantee, a protocol must have a $O(n \log n)$ **worst case complexity**.

3.3 Exercises

1. Apply the *All The Way* and *As Far* algorithms on this graph.



2. Try to simulate the algorithm *As Far* with node 4 as initiator on this graph. Repeat it with *All The Way*.



4 Synchronous Computations

4.1 Fully Synchronous Systems

In the distributed computing environments we have considered so far, we have not made any assumption about **time**. In fact, from the model, we know only that in absence of failure, a message transmitted by an entity will eventually arrive to its neighbor: the *Finite Delays* axiom. Nothing else is specified, so we do not know for example how much time will a communication take. In our environment, each entity is endowed with a local clock; still no assumption is made on the functioning of these clocks, their rate, and how they relate to each other or to communication delays.
For these reasons, the distributed computing environments described by the basic model are commonly referred to as **fully asynchronous systems**. They represent one extreme in the spectrum of message-passing systems with respect to time.
As soon as we add temporal restrictions, making assumptions on the the local clocks and/or communication delays, we describe different systems within this spectrum.
At the other extreme are fully **synchronous systems**, distributed computing environments where there are **strong assumptions** both on the **local clocks** and on **communication delays**. These systems are defined by the following two restrictions about time: *Synchronized Clocks* and *Bounded Transmission Delays*.

- **Synchronized Clocks**: All local clocks are incremented by one unit simultaneously. In other words, all local clocks 'tick' simultaneously. Notice that this assumption does not mean that the clocks have the same value, but just that their value is incremented at the same time;
- **Bounded Communication Delays**: There exists a known upper bound on the communication delays experienced by a message in absence of failures. In other words, there is a constant Δ such that in absence of failures, every message sent at time T will arrive and be processed by time $T + \Delta$. In terms of clock ticks, this means that in absence of failures, every message sent at local clock tick t will arrive and be processed by clock tick $t + \lceil \frac{\Delta}{\delta} \rceil$ (sender's time).

4.1.1 Overcoming Transmission Costs: 2-bit Communication

In order to overcome the transmission costs, we can exploit the following property: *Any information can be transmitted using 2 bits.* This property can be obtained as follows: suppose A wants to send a value X to B.

1. A sends a bit at time t (*Start counting*);
2. B receives at time $t_1 = t + 1$;
3. A waits X units of time;
4. A sends another bit at time $t + X$ (*Stop counting*);
5. B receives the second bit at time $t_2 = t + X + 1$.

In this case, $X = t_2 - t_1$, and we use:

- 2 bits;
- X units of time.

Can we improve this protocol?

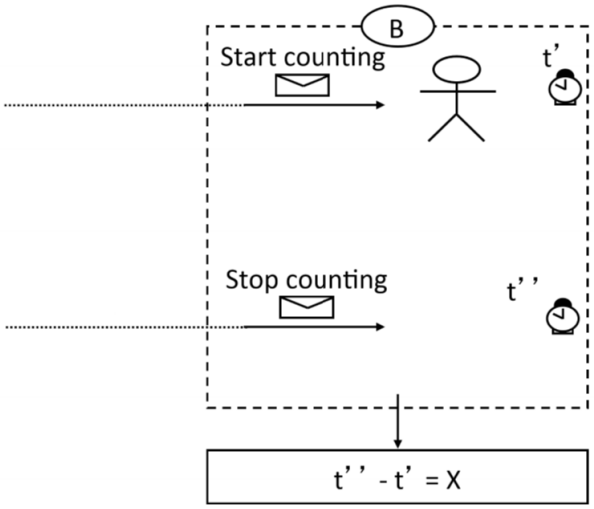


Figure 33: *2-bits communication protocol.*

1. A sends a bit b_0 at time t (*Start counting*);
2. B receives b_0 at time $t_1 = t + 1$;
3. A waits $\lceil X/2 \rceil$ units of time;
4. A sends another bit b_1 at time $t + \lceil X/2 \rceil$ (*Stop counting*), where $b_1 = \begin{cases} 0 & \text{if } X \text{ is even} \\ 1 & \text{if } X \text{ is odd} \end{cases}$
5. B receives the second bit at time $t_2 = t + \lceil X/2 \rceil + 1$.

In this case, $X = 2(t_2 - t_1) + b_1$, and we use:

- 2 bits;
- $\lceil X/2 \rceil$ units of time.

Example. *Suppose $X = 7$, then:*

- $t_1 = 1$;
- $\lceil X/2 \rceil = 3$;
- $b_1 = 1$;
- $t_2 = t_1 + 3$;
- $X = 2(3) + 1 = 7$

Again, we could improve the protocol as follows:

1. A sends a bit b_1 at time t (*Start counting*), where $b_1 = \begin{cases} 0 & \text{if } X \text{ is even} \\ 1 & \text{if } X \text{ is odd} \end{cases}$
2. B receives b_1 at time $t_1 = t + 1$;
3. A waits $y = \lceil X/4 \rceil$ units of time;

4. A sends another bit b_2 at time $t+y$ (*Stop counting*), where $b_2 = \begin{cases} 0 & \text{if } \lceil X/2 \rceil \text{ is even} \\ 1 & \text{if } \lceil X/2 \rceil \text{ is odd} \end{cases}$
5. B receives the second bit b_2 at time $t_2 = t + y + 1$.

In this case, $X = 2(2(t_2 - t_1) + b_2) + b_1$, and we use:

- 2 bits;
- $\lceil X/4 \rceil$ units of time.

4.1.2 Overcoming Transmission Costs: 3-bit Communication

In this case the idea is the following. For example, if we want to communicate $X = 40$ with 3

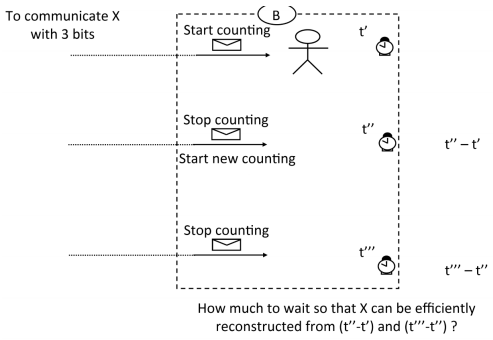


Figure 34: *3-bit communicators.*

bits, we can follow the protocol. The receiver, to decode the information, must compute

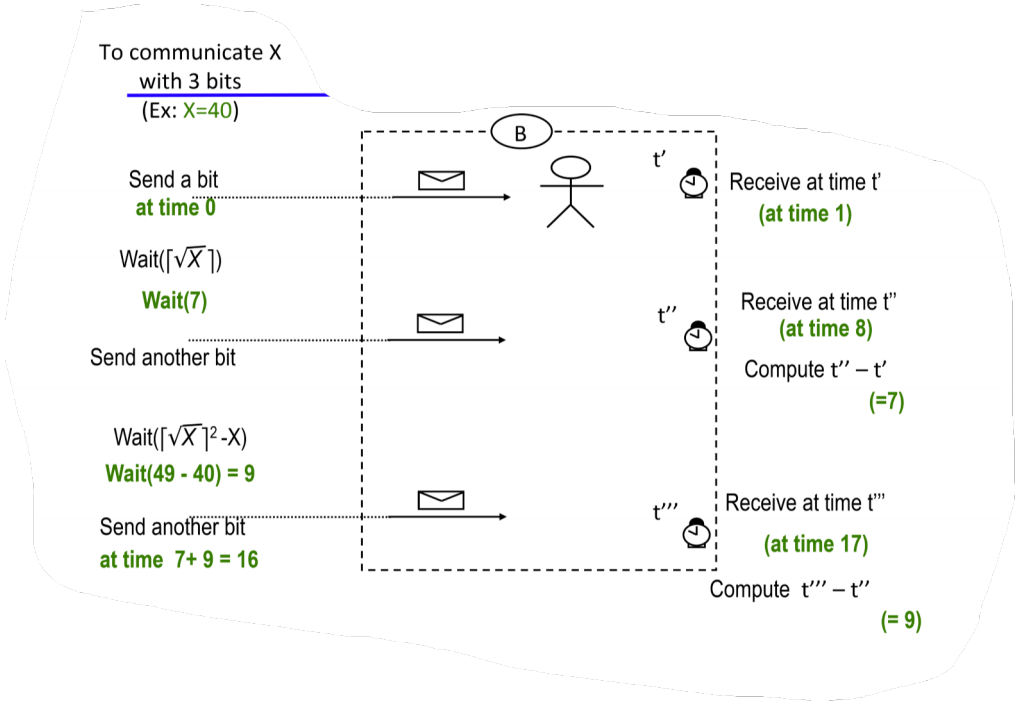


Figure 35: *Example.*

$$(t'' - t')^2 - (t''' - t'')$$

In the example, $7^2 - 9 = 49 - 9 = 40$. In this case we use:

- 3 bits;
- $O(\lceil\sqrt{X}\rceil)$

4.1.3 Overcoming Transmission Costs: k-bit Communication

In general, using k bits we use:

- k bits;
- $O(\lceil kX^{1/k}\rceil)$ units of time.

4.1.4 Pipeline

With communicators we have addressed the problem of communicating information between two neighboring entities. What happens if the two entities involved, the sender and the receiver, are not neighbors? Clearly the information from the sender x can still reach the receiver y, but other entities must be involved in this communication. Typically there will be a chain of entities, with the sender and the receiver at each end; this chain is, for example, the shortest path between them.
If we use communicators, we have the following situations.

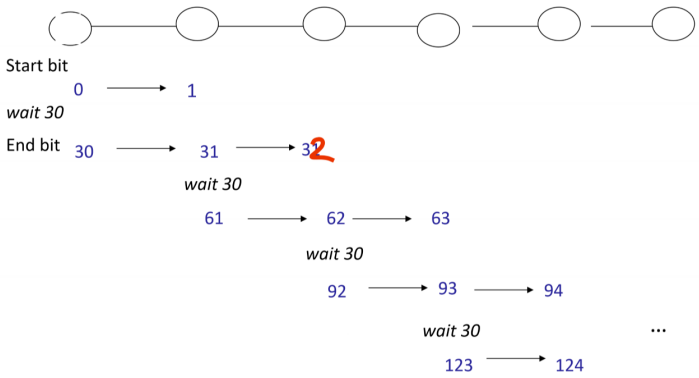


Figure 36: *Multiple communicators.*

On the other hand, we have pipelines.

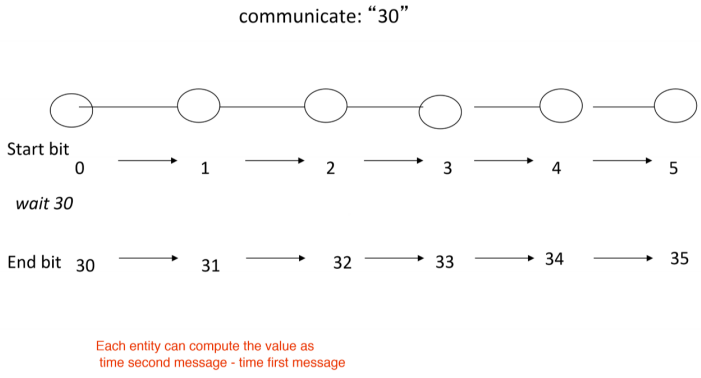


Figure 37: *Pipeline.*

Pipeline can be exploited also for communicating the maximum value.

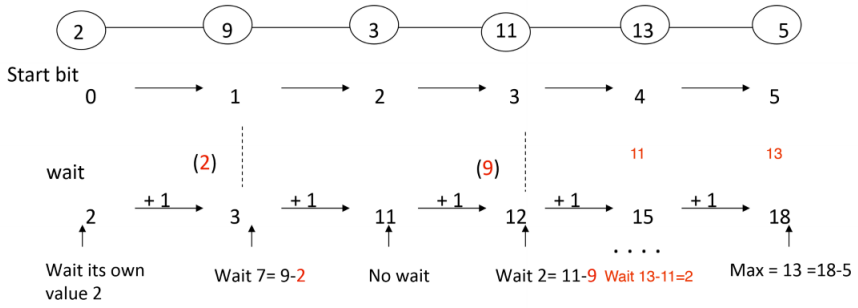


Figure 38: *Pipeline: communicating the maximum*

4.2 Min-finding and election

We now describe the **Speeding** algorithm, for minimum finding and election. The general idea is that messages travel at different speeds. In this case:

- The **knowledge** of n is **not necessary**;
- We consider a **synchronous** and **unidirectional** version;
- We assume **simultaneous start**, but it is not necessary.

We have two ways of eliminating IDs:

1. Like in *AsFar*, large IDs are stopped by smaller IDs;
2. Small IDs travel faster so to catch up with larger IDs and eliminate them.

In particular, each message travels at a speed which depends on the identity it contains, i.e. identity i travels at some speed $f(i)$. Since speed is assumed to be unitary, and the same for every message, how can we change it? We introduce appropriate delays.

1. When a node receives a message containing i it waits $f(i)$ (e.g. 2^i) ticks;
2. When a node receives its own id, it becomes the leader and sends a notification message around the ring. This message will not be delayed.

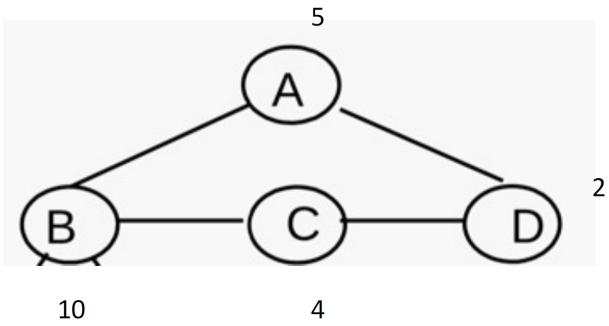
If we use $f(i) = 2^i$, in time $2^i n + n$ the smallest id i traverses the ring. Let the second smallest be $i + 1$, with waiting time 2^{i+1}: how many links does it have the time to traverse (at most) while the smallest ID goes around? The answer is $\frac{2^i n+n}{2^{i+1}} \approx \frac{n}{2}$ links. See the example of L19.
In this case, we use:

- $O(n)$ messages;
- $O(n \log ID)$ bits, where ID is the largest ID;
- $O(2^i n)$ units of time, where i is the smallest ID.

4.3 Exercises

1. Execute the variants of the 2-bits communication protocol for $X = 7$;
2. Execute the 3-bits communication protocol for $X = 23$. Solution on slide 12 of L19;
3. Given two entities, Alice and Bob, in a synchronous system, how can Alice communicate number 59 to Bob using 3 bits (3 bit communicator)?
4. Execute the Pipeline algorithm both for communicating the message $X = 13$ and for communicating the maximum value;

5. What is the Speeding algorithm in a synchronous ring? Describe it (with code, pseudocode, words, as long a it is complete, precise and clear). Provide an execution example on this graph.



5 Peer-to-Peer systems

Peer-to-Peer systems (**P2P**) belong to distributed systems, which differ from centralized systems from the fact that the communication is characterized by message exchange, while in centralized systems a shared memory is responsible for communication.

5.1 Architecture

P2P can be seen as an organizational principle, in which the participating **entities are all equal** (in principal), in contrast with client-server systems, where each entity has a clear role:

- The **server** provides a service and manages the resources;
- The **client** uses the service.

Thus, in P2P everyone acts as client and as server at the same time.
In general, P2P systems can be classified as:

- **Hybrid**, e.g. *Napster* and *KaZaA* (super peer);
- **Pure**, which in turns can be classified as:
  - **Structured**, e.g. *Chord*;
  - **Unstructured**, e.g. *Gnutella*.

5.2 Concepts and properties

The basic **concepts** of P2P are:

- **Self organizing** (no central management);
- Based on **voluntary** (i.e. not forced) collaboration;
- Peers are all **equal** (more or less);
- **Large number** of peers in network;
- High **autonomy** from central servers;
- Exploits **resource** (e.g., storage) of every peer in the network;
- Peers join and leave the network, so it is a very **dynamic network**.

On the other hand, the typical **properties** of P2P are:

- Unreliable;
- Unmanaged;
- Uncoordinated;
- Resilient to attacks;
- Large collection of resources.

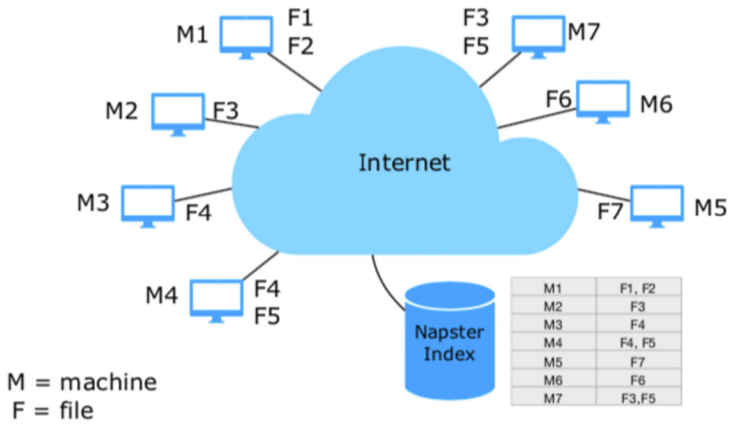


Figure 39: *Centralized Napster Index.*

5.3 Search

5.3.1 Search in hybrid P2P systems

Napster The **idea** behind this system is the following: share content, storage and bandwidth of individual (home) users. In particular, each user machine stores a subset of files and can download files from all users in the system, using the *Centralized Napster Index*, which stores the system index that maps files to alive machines (look-up table).
In this sense:

- The peers send **metadata** to the look-up server;
- When a resource request arrives, the server returns the list of the peers that store the resource;
- **Data** are then exchanged among peers

The main **challenges** are:

- *Where is a specific file stored?*
- *How to scale the system up to thousands or millions of machines?*
- *How to handle dynamicity (machines coming and going)?*

Among the **advantages**, we can underline:

- **Easy** to implement sophisticated search engine technique for centralized index;
- **Ideal** (user) **machine load**;
- Central servers have a **complete** and **consistent view** of the system (who is aware of which files are available);
- Answer always **correct** (no, means there is no file).

On the other hand, among the **disadvantages**, we have:

- **Scalability**: Centralized index has to handle all the queries;
- **Robustness**: Centralized index is single point of failure (failures, attacks...): if the index is not available, the system does not work;
- Result **unreliable**: no guarantee about file content and correctness of user information.

5.3.2 Search in pure unstructured P2P systems

Gnutella In this case we have a **pure** P2P system, without a central server: Peers have an initial a set of addresses known for the first connection, and are equally treated, no matter which bandwidth they have and how many files they share. Each peer provides both the files and sends/replies to routing requests, and each peer is both a client and a server. In general, this system is hard to control/regulate.

- Each peer "knows" a **subset of neighbouring peers**, and there are some **cache servers** that maintain as many peer addresses as possible. When the application starts, it contacts one of these cache servers that will add the new peer to the P2P network;
- Requests from a peer R are sent from neighbour to neighbour ($PING$): The message stops either when the resource is found or after a limited and predefined number of steps (TTL, $Time\ To\ Live$);
- If the resource has been found, the address of the peer P that stores it is sent to R (the ID of R can be sent together with the request) ($PONG$). R will directly contact P and will download the resource

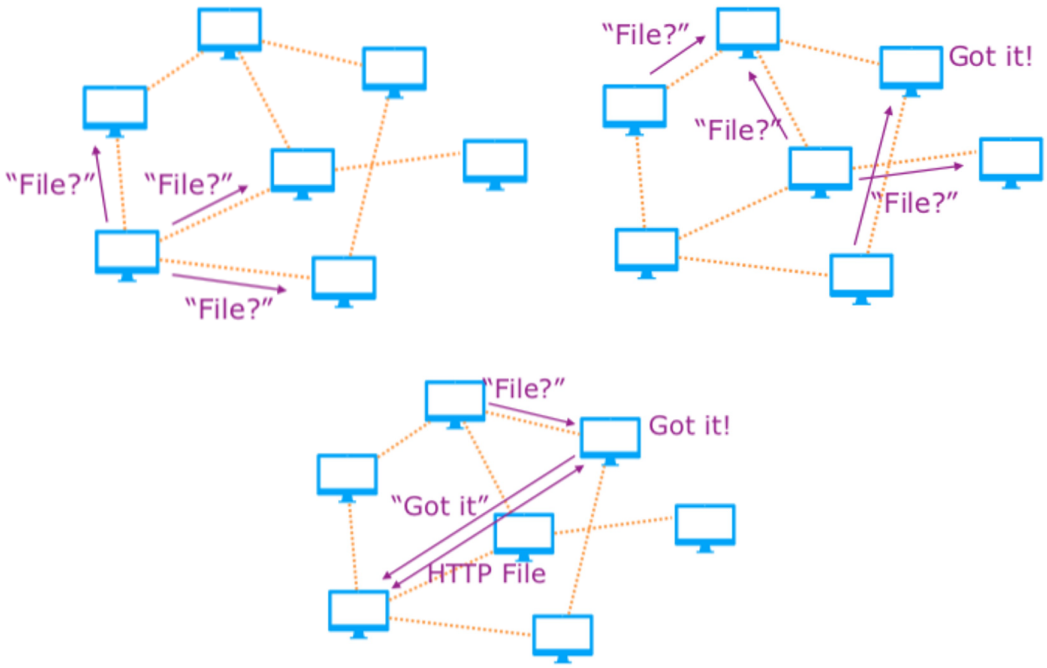


Figure 40: *Example.*

Among the **advantages**, we have:

- This system is completely **distributed** and **decentralized**;
- It is **robust** w.r.t. randomized node failures.

On the other hand, the **disadvantages** are:

- R might receive **redundant results** (if the resource is in many different peers), or nothing although the resource is stored by a far away peer (TTL too small);

- One **peer** might be **visited many times** (e.g., A from F and G);
- Increasing the TTL increases the **resource availability**, but also the flood of the network increases;
- Risk of **"Denial of service" attacks**, if somebody floods the network with resources requests. Thus, we need to maintain resource statistics and close the network to "offending peers".

Iterative Deepeinig Iterative deepening represents another example of pure unstructured P2P system. In this case the search is implemented as follows:

1. The system is **flooded** with a limited TTL: If the resource is not found we start with a bigger TTL (predefined sequence of TTLs);
2. We **repeat** up to when the resource is found or when a boundary TTL is found.

5.3.3 Search in hybrid systems

Super KaZaA In this case the idea is to organize the machines using a hierarchy, so there are two kinds of nodes in a two-tier hierarchy:

- **Ordinary nodes** (**ON**) at the lower-level tier, which is a normal machine run by the user. An ordinary node belongs to a single supernode;
- **Supernodes** (**SN**) at top-level tier, which is a machine run by a user with more resources and responsibilities. It acts like a Napster centralized index for its ON. The super peers also operate as normal peers and exchange information directly, but they do not have information about ONs belonged by other SNs.

In this sense, this system combines the characteristics of hybrid and pure systems. Finally, in order to join the system, an ON sends request to one SN (its IP address known somehow) and sends list of files to share. In general, we also need mechanisms to handle SNs.
In order to **search** a file:

1. ON sends request to its SN;
2. The SN answers for its ONs and forwards request to other SNs;
3. Other SNs answer for their own ONs.

Clearly, we have may consider several questions, e.g. *What is a good number of leaves for each super peer? How should super peers connect together (structured or not)?*.
The main **advantage** of this system is that it combines advantages from Napster and Gnutella, providing an **efficient searching** within SN *Flooding* restricted to SNs.
On the other hand, the **disadvantages** are:

1. Combine **disadvantages** from Napster and Gnutella, i.e. an existing file might not be found;
2. SNs are **points of failure**.

5.3.4 Search in structured systems

We will consider *Chord* lately in the sections.

5.3.5 Other applications

- E-Donkey: similar to Napster but with different servers;
- E-mule (for windows) and A-mule (Mac, Linux): open source, uses the E-Donkey e Kad networks;
- Bit torrent. In this case:
  - Many users can simultaneously download the same file without too much delay;
  - Files are not downloaded from the same server but they are divided into different portions, so the same file is distributed among many peers;
  - Each peer that makes a request, offers already downloaded portions to the other peers, thus contributing to the downloading of the other peers;
  - Works in Microsoft Windows, Mac OS, Linux and Android.

5.3.6 Comparison

Category		Hybrid	Unstructured	Structured
		Blindly	Informed	
Example	Napster	Gnutella	Int. BFS	Chord
Structure	Central.	Random	Random	Fixed
Search	Index Server	Flood	Opt Flood	DHT
Info Data	deterministic	Nothing	Partial	High Prob.
Data Loc.	Everywhere	Everywhere	Everywhere	Fixed

Figure 41: *Comparison of the systems.*

5.4 Distributed Hash Tables

We now consider the following questions: *Where to store the information in a P2P system? How to find it?* In this sense, we need to consider the following parameters:

- **System scalability**: limit the communication overhead and the memory used by the nodes with respect to the number of nodes in the system;
- **Robustness** and **adaptability**: in the presence of failures and changes.

We may consider:

- **Centralized** approach: we send a request to the server, so we need $O(N)$ memory on the server, while search is performed in $O(1)$ step (to reach the server);
- **Decentralized** approach: we send a request to all our neighbours (different optimizations, e.g. TTL), so we need $O(1)$ memory, while search is performed in $O(N^2)$ steps,

Can we find a solution that finds a good memory/step trade-off? The answer is given by **Distributed Hash Tables** (**DHT**), which are characterized by:

- $O(\log N)$ steps to find the information (Time);
- $O(\log N)$ entries in the routing table of each node (Space).

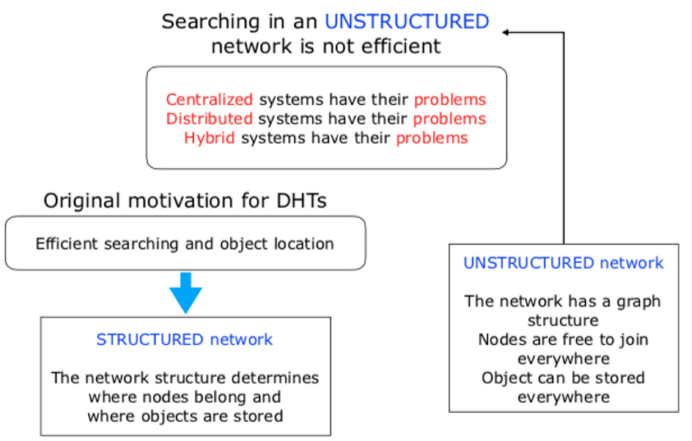


Figure 42: *Why Distributed Hash Tables?*

Moreover, this technique is also characterized by **adaptability**: it is simple to insert (assign the information) and to remove nodes (reassignment to the neighbouring nodes), and it **balances information** on the nodes (makes routing efficient).
In general, peers and data are mapped in the same address space through hash functions, and:

- For the **nodes** we consider the **hash** of the **IP**;
- For **data** we consider the **hash** of the **content** (title, etc..).

The DHT are used by many different systems (Chord, Can, Pastry, Tapestry,..), and all these systems use APIs such as: *Put(key,value)*, *Get(key)* and *Value*.

5.4.1 Chord

In *Chord*:

- **Data** are distributed among **peers** using a precise algorithm;
- **Data** are replicated to improve **availability**.

For the **assignment**:

- Each peer has an **ID** (hash of the IP);
- Each resource has a **key** (hash of the title, etc.);
- The **assignment** uses *SHA-1* (Secure Hash Standard), which is now deprecated, and uses long keys to avoid collisions (same key), e.g. 160 bit;
- The peer stores resources with keys similar to the one of the peers (same logical addressing space);
- Given a key of a resource the peer sends the request to the peer with the most similar key.

Finally, through **consistent hashing**:

- We assign the keys to the peers;
- There should be load balancing so that with very high probability each peer receives the same number of resources/keys;
- We ask for key updates when a peer connects/disconnects from the network;

- The insertion of the N-th key with high probability will require the movement of $1/N$ other keys.

The **advantages** of this system are:

- We maintain **consisting hashing** although the information is not stored in all the nodes;
- The system is **simple**, correctness and performance are easy to prove;
- It takes at most $O(\log N)$ to reach the destination;
- A peer requires a table of $O(\log N)$ bits for an efficient routing, but the performance decreases when the tables are not updated;
- The insertion/removal of a node generates $O(\log^2 N)$ messages.

The **logical space** is a circular ring $0, .., 2^m - 1$, ring (mod 2^m): A resource with key K is stored in the node which is the closest successor from K (*successor(K)*).

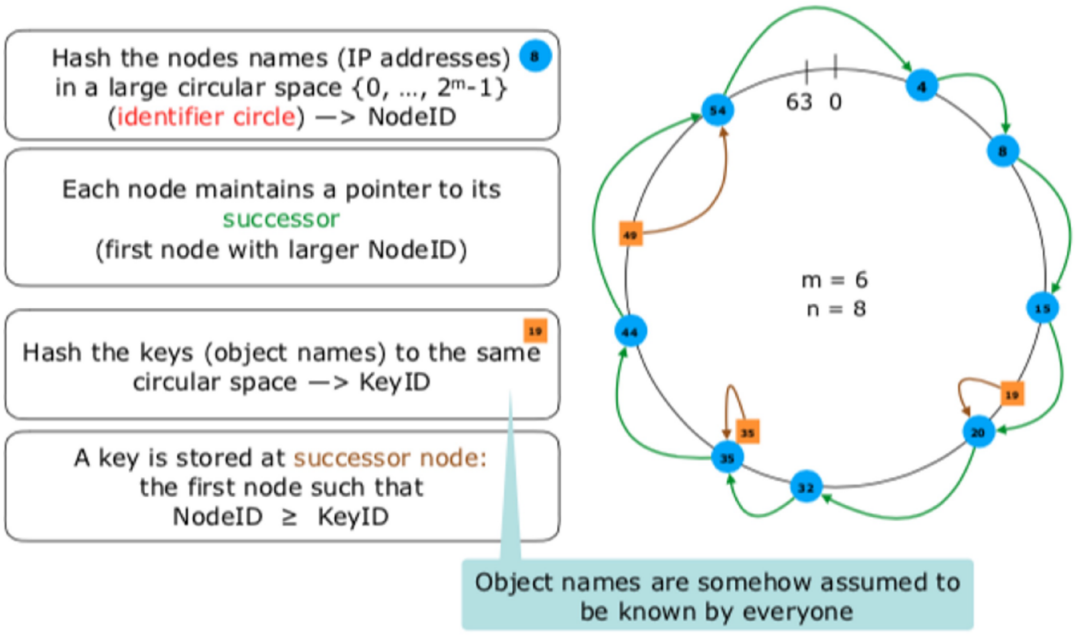


Figure 43: *Chord logical space.*

Notice that two nodes cannot have the same hash value, and if a peer crashes, the information could be not precise and correct.

Example. *Suppose that $m = 7$, thus the ring goes from 0 up to $2^m - 1 = 127$. Notice that each key is stored at its successor, which is represented by the node with the next higher ID. For example, the successor of peer 22 is 87, of 87 is 103, of 103 is 22. The predecessor is the active peer the precedes (in clockwise direction) in the ring, e.g. the predecessor of 87 is 22. Thus, keys are not necessarily consecutive.*

Routing. First version Routing represents a **simple** algorithm: each node stores only its successor: if the resource is not on this node, then the query is sent to the successor.
The algorithm is characterized by:

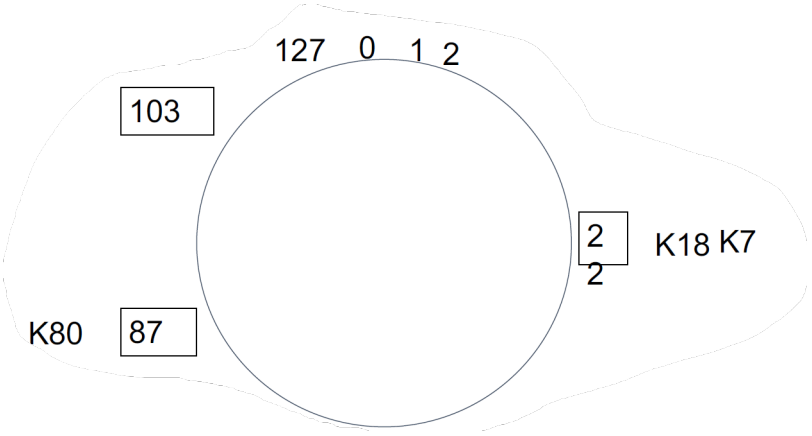


Figure 44: *Example.*

- $O(1)$ memory;
- $O(N)$ search: in the worst case we have to visit the whole ring before we find the resource;
- The failure of a node blocks the procedure.

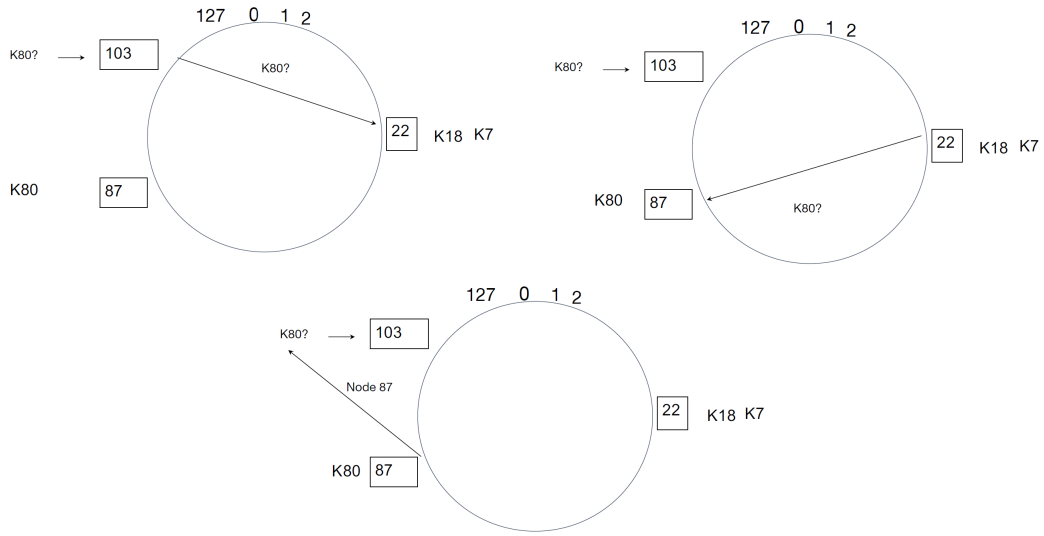


Figure 45: *Example of Routing algorithm.*

The main **limit** of this version of the algorithm is that in the worst case it requires **too many operations**, so the idea is to maintain other routing information. Note that, this extra information is not required for the correctness of the procedure but it helps speeding up the search. The protocol works given that the value of the next successor is correctly maintained/updated.

Routing. Second version In this case each node stores N successors: if a resource K is not on a node, the node looks in the successor node.
In this case, we have:

- $O(N)$ memory;
- $O(1)$ search.

However, the best solution is in the middle.

Routing. Third version Each node stores m values. The search of K is sent to the **furthest known predecessor** of K. We store more values of close nodes and less of more distant nodes, thus routing is more precise close to a node.
In this case, we have:

- $O(\log N)$ memory;
- $O(\log N)$ search.

Each node maintains:

- A **finger table**, where the i-th entry of the finger table of x is the first node that succeeds or equals $(x + 2^i) \mod 2^m$;
- **Predecessor node**.

An item identified by *id* is stored on the successor node of *id*.

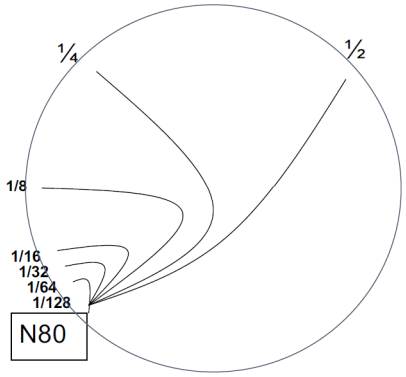


Figure 46: *Routing: finger table.*

Example. *We assume* $m = 3$*, thus* $2^m = 8$*. If node* $n_1 : (1)$ *joins, all entries in its finger table are initialized to itself (no other peer is there). The entries are* $1 + 2^0 = 2$*,* $1 + 2^1 = 3$ *and* $1 + 2^2 = 5$*. If node* $n_2 : (2)$ *joins, the situations becomes the following. Now, nodes* $n_3 : (0)$ *and*

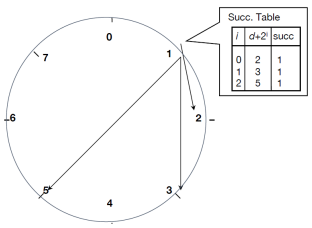


Figure 47: *Example #1.*

$n_4 : (6)$ *join. Now, suppose that the items are* $7 : (0)$ *and* $1 : (1)$*, if we get a query for item* 7*, we have the following situation. As we said, upon receiving a query for item ID, a node:*

- *Checks whether it locally stores the item;*
- *If not, it forwards the query to the largest node in its successor table that does not exceed id.*

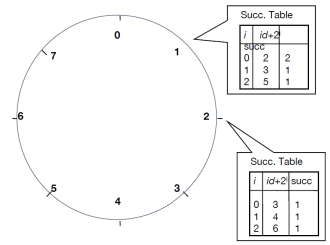


Figure 48: *Example #2.*

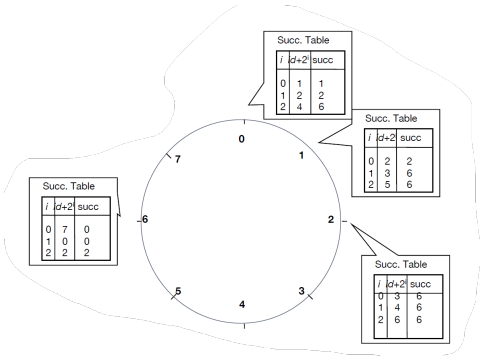


Figure 49: *Example #3.*

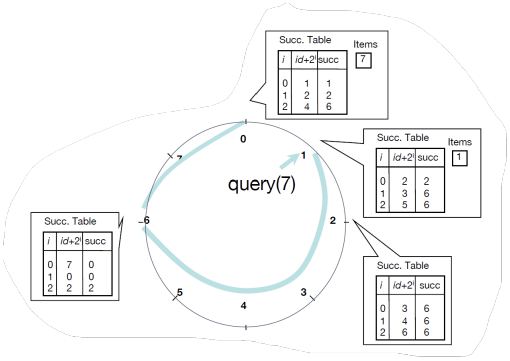


Figure 50: *Example #4.*

In order to compute the **complexity** of the algorithm, we need to consider the following theorems:

- In a CHORD network with m-bits ids, the number of nodes that has to be traversed for the routing of a single query is at most m;
- Given a query q the number of nodes that has to be traversed to find the successor of q in a CHORD ring of N nodes is, with high probability, $O(\log N)$;
- The finger table of a node x contains at most $O(\log N)$ distinct entities.

CHORD can deal with the **dynamical changes** of the network such as:

- Node failures (replica are needed);
- Network failures;
- Adding of new peers (insertion);

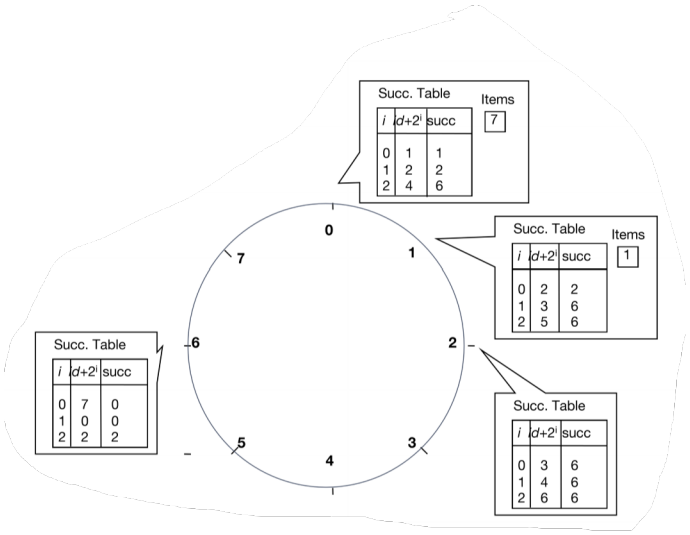
- Removal of a peer

Two **operations** are required:

- Successor update (routing correctness);
- Finger table update (routing efficiency).

5.5 Exercises

1. Implement the search for item 1 by peer 6, and build the table for $n_5 : (4)$.



6 Mobile Agents

6.1 The model

In this case the model is represented by a graph $G = (V, E)$, where the nodes are the **hosts**, and the edges are the **communication links**. Moreover, each edge is characterized by an edge label, called **port number**, and all the edge labels are distinct.
For what regards the mobile agents, they:

- Have an **homebase**;
- Have **computing capabilities**;
- Have **local storage**;
- Can **move** from a node to a neighboring one;
- Have the **same behaviour**, i.e. they execute the same protocol;
- **Communicate**, e.g., through whiteboards (using a storage of $\log n$ tipically), stored in nodes and accessed in mutual exclusion (etc..)

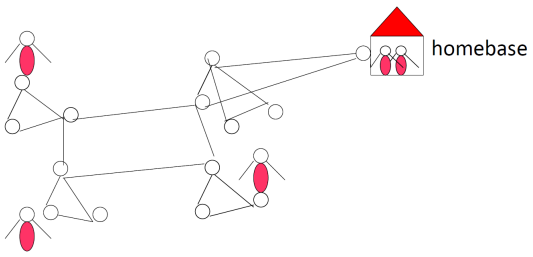


Figure 51: *Mobile agents.*

Moreover, we can have:

- **Asynchronous** setting (i.e., actions take a finite but unpredictable amount of time);
- **Synchronous** setting (i.e., an agent traverses an edge in one unit of time).

Notice that in all the possible investigations (e.g. *exploration*, *map construction* etc..), it is assumed that the network is **safe**. However, unfortunately networks (such as the Internet) can be **dangerous**.

6.2 Black hole search

In general, we can distinguish:

- **Harmful Host** (harmful stationary process): exists also in regulated systems where agents cooperate (hardware or software failure);
- **Harmful Agent** (malicious mobile process): acute in unregulated non-cooperative settings (e.g. Internet).

If we're dealing with harmful hosts, the main task is to protect the agents.
A very famous example of harmful host is the **black hole**. This host:

- **Destroys** any agent arriving at that node;
- Does not leave any **trace** of destruction;

- Its **location** is **unknown** to the agents.

In this sense, in this case the goal is to **find** and **report** the **location of the black hole**, so that no agent is destroyed. From this we can derive that at least one agent must survive and know the location of the black hole.
Notice that the cost and efficiency of the algorithms we will consider is given by:

- **Size of the Team** (Number of Agents): the agents are required to locate the black hole, so our goal is to minimize this quantity;
- **Cost** (Number of Moves);
- **Time**.

6.2.1 Cautious Walks

The idea of cautious walk is that we want at most one agent to walk on a dangerous link. In particular, each port can be:

- **Unexplored** i.e. no agent traversed, so it could be dangerous;

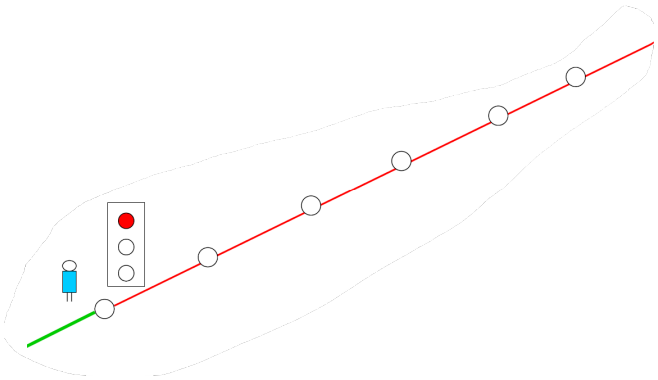


Figure 52: *Unexplored port.*

- **Active**, i.e. an agent is traversing it;

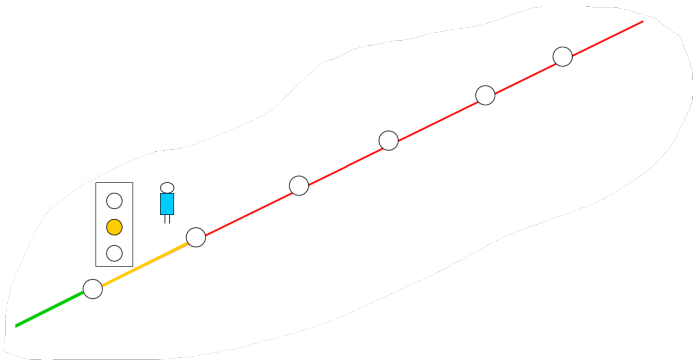


Figure 53: *Active port.*

- **Explored**, i.e. an agent traversed the edge and successfully returned.

The main **disadvantage** of this solution is that **asynchrony** makes this problem difficult, since we cannot distinguish between a slow agent and one that disappeared in the black hole. This

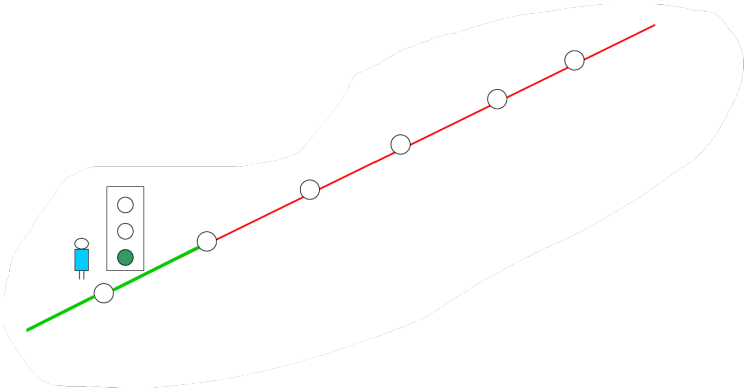


Figure 54: *Explored port.*

becomes **impossible** if n is unknown, and in general if the node connectivity of the graph G is lower than 2 (intuitively, we need at least two paths to reach each node). This implies that in such conditions it is impossible to verify if a black hole exists.

6.3 Ring

In this case, we have:

- n nodes: each node has two ports, left and right. Notice that n is known;
- k agents;
- A single black hole.

Clearly, one agent cannot locate the black hole alone. *What about two agents?* In this case the agents, starting from the home base, must explore using **cautious walk**. The agents must explore disjoint areas otherwise they could both disappear!

Example. *Suppose that in this case the agent on the left goes faster than the agent on the right. Now, the agent on the right get destroyed by the black hole, while the other one reaches the last*

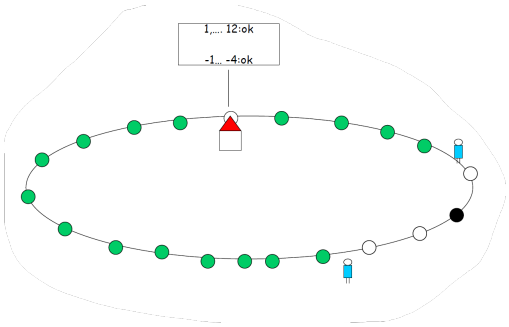


Figure 55: *Example #1.*

host. Since n is known, and since the agent crossed $i = 14$ hosts on one side, while the other crossed $n - i - 2 = 5$ on the other, it is able to locate the black hole.

In this case the cost of the algorithm is $O(n^2)$.

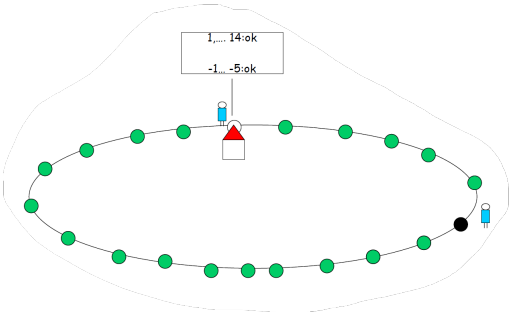


Figure 56: *Example #2.*

6.4 Intruder capture

As we said before, in our graph we could also have **harmful agents** (or **intruders**): in this case, the task becomes to protect the hosts from such agents. In this case, the intruder:

- **Moves** from a node to a neighboring one;
- **Moves** arbitrarily fast;
- **Cannot cross** a node guarded by an agent;
- It is **invisible** to the agents;
- Can permanently **see** the position of the other agents.

Considering the intruder capture problem:

- **Initially**, the agents are located at the **homebase** and form a team;

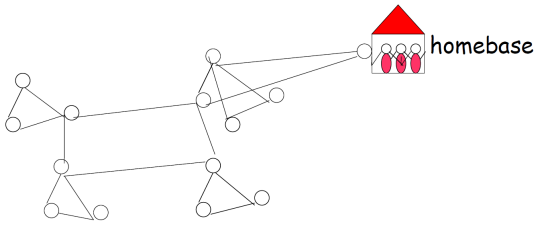


Figure 57: *Intruder problem: initial situation.*

- At the **end**, the agents **capture** (surround) the **intruder**.

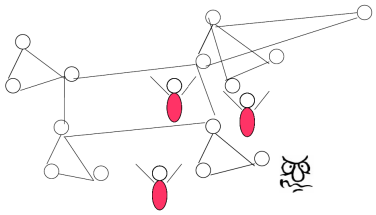


Figure 58: *Intruder problem: final situation.*

We can say that the intruder capture problem is equivalent to the decontamination problem. In this case:

- Initially, the agents are located at the homebase and form a team. The **whole network is contaminated** (except the homebase);
- An agent cleans a node when it enters it;
- At the end, the **whole network** must be **clean**;
- A node becomes contaminated if it is not protected by an agent and at least one of its neighbours is contaminated.

Similarly, we can consider the edge **decontamination problem**:

- Initially, the agents are located at the homebase and form a team. The nodes and edges of the network are contaminated (except the homebase);
- An agent cleans an edge when it traverses it;
- At the end all the nodes and edges of the network must be clean.

In order to solve such problem, we need to consider contiguous monotone strategies:

- **Contiguous**: agents move only to neighbouring nodes;
- **Monotone**: no recontamination can occur.

The idea of such a strategy is to have a frontier of agents, which moves towards the contaminated area. In this case the complexity is computed considering:

- **Number** of **agents**;
- **Number** of **moves**;
- **Time**: synchronous or asynchronous (ideal time);
- **Memory** of agents and nodes (whiteboard).

6.5 Results

In the following results we consider these assumptions:

- **Visibility**: agents can see the state of their neighbours (*clean*, *contaminated*, *guarded*);
- **Locality**: agents have only local knowledge.

6.5.1 Decontamitaning a mesh

We consider an asynchronous system, where the storage for a node and an agent requires $O(\log n)$ bits, and we'll focus on both visibility and local models on a $m \times n$ mesh (with $m \leq n$).
The strategies are:

- Strategy 1: **With a Synchronizer** (local model). In this case the searching agents do not have visibility power (i.e., they cannot see their neighboring nodes). The synchronizer is an agent that coordinates (or synchronizes) the moves of the other agents;
- Strategy 2: **Agents with Visibility** (visibility model). Visibility power means agents can see their neighboring nodes. The agents move independently and the synchronizer is not required.

The main idea is the following:

1. **Start** from the **homebase**, and contiguously **clean the contaminated network** by maintaining a vertical barrier of agents (to avoid recontamination), which has to work asynchronously;
2. Move **one column** at the **time**.

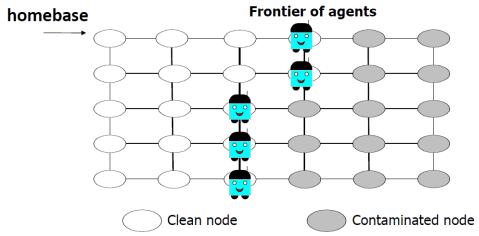


Figure 59: *The frontier of agents.*

Node search with synchronizer In this case we work with a $m \times n$ mesh, and the strategy is the following:

1. We have $m + 1$ agents (m agents and 1 synchronizer);
2. **Initialization phase**: place the agents in the first column. This phase requires:

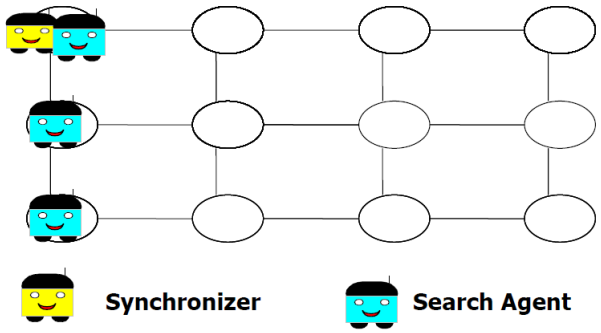


Figure 60: *Initialization phase.*

- $m - 1$ time steps;
- $\frac{m(m-1)}{2}$ moves.

3. **Cleaning phase**: the synchronizer moves the agents one column at a time. This phase

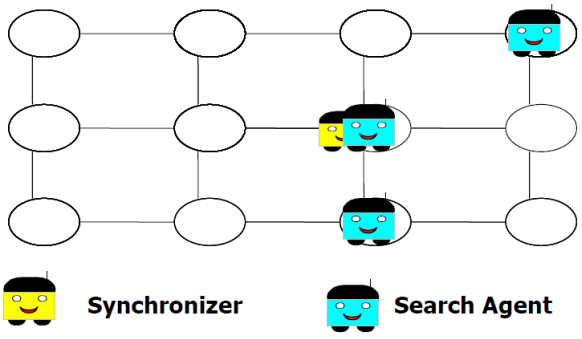


Figure 61: *Cleaning phase.*

requires:

- $m - 1$ time units NORTH and SOUTH on $n - 1$ columns, i.e. $(m - 1)(n - 1)$ time units;

- $n-2$ time units EAST to the next column;
- 1 move EAST by the last move;
- $n-1$ moves EAST by the m nodes, so $m(n-1)$;
- $n-2$ moves EAST by the synchronizer, which also moves $n-1$ SOUTH and NORTH for $m-1$ moves.

Thus, in total, we have:

- $m+1$ agents
- $mn-2$ time units;
- $\frac{m^2+4mn-5m-2}{2}$ moves.

Theorem. *The algorithm "Search Synch" is* ***correct****, i.e. the mesh is decontaminated and the cleaning is contiguous and monotone.*

Edge search with synchronizer We have seen how a synchonizer can decontaminate the nodes, what about the edges? The solution is quite simple: at the end of the cleaning phase, the synchronizer cleans the edges of the **last column**.

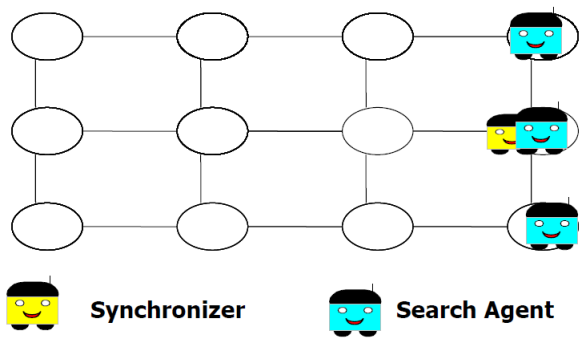


Figure 62: *Cleaning phase in edge search*

Thus, we need an extra m moves and m time units.

Node search with visibility Now we consider the case where the agents see the neighboring nodes. In particular, each searcher s has a local variable WB:

- If WB = empty, s writes *clean*, guards the node and moves EAST when all neighbours except the one EAST are *clean* or *guarded*;
- If WB = clean, move SOUTH.

This strategy is implemented so that the only uncleaned node is the one in front of the barrier. In this case, we have:

- m agents.
- $m+n-2$ time units;
- $\frac{m^2+2mn-3m}{2}$ moves.

Theorem. *The algorithm "Node Search Visib" is* ***correct****, i.e., the mesh is decontaminated and the cleaning is contiguous and monotone.*

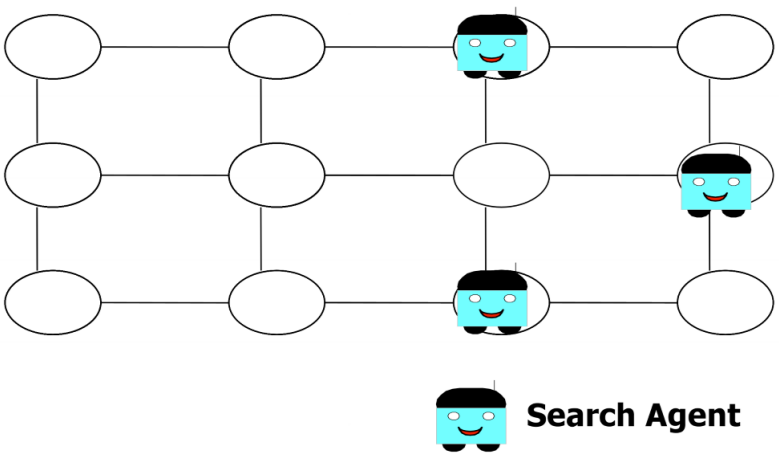


Figure 63: *Node search with visibility.*

Lower bound In any $m \times n$ Mesh ($m \leq n$), the solution of the node search problem requires at least:

- m agents
- $n + m - 2$ time units: initially all at the homebase, upper left corner. Then, at least one agent has to reach the node in the bottom right corner, i.e., at least $n + m - 2$ time units;
- nm moves: initially all at the homebase, upper left corner. Then, all nodes (nm) have to be visited by some agent (the homebase is already clean);

Search in a $m \times n$-Mesh				
Model	Number Of	*Lower Bounds*	*With a Synchronizer*	*Agents with Visibility*
Node	Agents	m	$m+1$	m
	Moves	mn	$\frac{m^2+4mn-5m-2}{2}$	$\frac{m^2+2mn-3m}{2}$
	Time	$m+n-2$	$mn-2$	$m+n-2$
Edge	Agents	m	$m+1$	$m+1$
	Moves	mn	$\frac{m^2+4mn-3m-2}{2}$	$\frac{m^2+4mn-5m}{2}$
	Time	$m+n-2$	$mn+m-2$	$mn-m$

Figure 64: *Complete results.*

7 Robots

A **robot** can be defined as a **physically-embodied**, **artificially intelligent** device with **sensing** and **actuation**. Among the properties of such devices, we have:

- A robot can **sense** and it can **act**;
- It must **think**, or process information, to connect sensing and action.

Is a washing machine a robot? Most people wouldn't say so, but it does have sensing, actuation and processing. In this sense, a possible distinction between appliance and robot (David Bisset) relies on whether the workspace is **physically inside** or **outside** the device. The **cognitive** ability required of a robot is much higher: the outside world is complex, and harder to understand and control.
The most widely-used robots today are **industrial robot** 'arms', mounted on fixed bases and used for instance in manufacturing. The task of a robot arm is to position an end-effector through which it interacts with its environment. These types of robots most operate in highly controlled environments.
There is a new wave of advanced mobile robots now aiming at much more flexible robots which can interact with the world in human-like ways. This is the current goal of significant research teams; e.g. *Willow Garage* and *Evolution Robotics* in the USA.

7.1 Oblivious Mobile Robots

The focus of our course will be given to **mobile robots**. A mobile robot needs actuation for **locomotion** and sensors for **guidance**. They are ideally **untethered** and **self-contained**: power source, sensing, processing on-board (Return to charging station? Off-board computing? Outside-in sensing?).
Required competences include:

- **Obstacle avoidance**;
- **Localisation**;
- **Mapping**;
- **Path planning**.

, as well as whatever specialised task the robot is actually trying to achieve!
The main applications of mobile robots are:

- **Field** Robotics
  - Exploration (planetary, undersea, polar);
  - Search and rescue (earthquake rescue; demining);
  - Mining and heavy transport; container handling;
  - Military (unmanned aircraft and submarines, insect robots).
- **Service** Robotics
  - Domestic (Vacuum cleaning, lawnmowing, laundry, clearing the table..?);
  - Medical (helping the elderly, hospital delivery, surgical robots);
  - Transport (Autonomous cars);
  - Entertainment (Sony AIBO, QRIO, Lego Mindstorms, Robocup competition, many others).

, whereas we may have some tasks, for example:

- **Basic coordination** task: gathering, specific patterns, alignment, scattering;
- **Complex** tasks: mine sweeping, hazardous retrieval, rescue operations.

7.1.1 Locomotion

In general, **wheels** are most common, in various configurations. **Legs** increase mobility, but with much extra complication. Robot **size** affects power requirements/efficiency, actuator specifications. The **pose** refers to both **position** and **orientation** together, more generally, we will talk about about a robot's state, which is a set of parameters describing all aspects of interest.

7.1.2 Sensing

Sensing is usually divided into two categories:

- **Proprioceptive** sensing, i.e. the 'self-sensing' of a robot's internal state;
- **External** sensing, of the world around a robot.

, although sometimes the distinction is not completely clear (e.g. a magnetic compass would normally be considered proprioceptive sensing).
Moreover, most mobile robots have various **sensors**, each specialised in certain tasks. Combining information from all of these is often called 'sensor fusion'. Sensors which measure a robot's internal state:

- Wheel **odometry** (encoders, or just checking voltage level and time);
- **Tilt** sensors (measure orientation relative to gravity);
- **Gyros** (measure angular velocity);
- **Compass**;
- **Internal force** sensors (for balance).

7.2 Computational model

We will consider **weak mobile robots**, with the following characteristics:

- **Autonomous** (no central control);
- **Homogeneous** (run same software);
- **Identical** (indistinguishable);
- **No communication capabilities**.

7.2.1 Capabilities

Each robot's basic capabilities refere to:

- **Processing** and **Storage**: limited, execute same protocol;
- **Sensorial**: "see" other robots, local coordinate system;
- **Motorial**: move towards a destination

7.2.2 Life cycle

Moreover, a robot's lifecycle is composed by the following states:

- **Look**, i.e. the robot uses its sensors to observe the world. The result of this phase is a snapshot of the external world;
- **Compute**, i.e. the robot receives in input the position of the other robots, and produces a destination point;
- **Move**, i.e. the robot moves towards the computed destination (it might not reach it);
- **Sleep**, i.e. the robot may be idle (e.g. to recharge battery).

7.2.3 Factors

Two important factors are **visibility** and the **level of agreement**. For what regards visibility, we have that it is limited by the radius of the robot, while we have different levels of agreement on a coordinate system:

- **Total** agreement (both axes and both directions);
- Agreement on **axes** and **one orientation**;
- Agreement on **axes**;
- **No** agreement.

7.2.4 Time and Synchronization

There are three basic models:

- **Fully synchronous** ($FSYNC$): in this case there is a **global clock** tick reaching all robots simultaneously. At each clock tick every robot becomes active and perform its cycle atomically;
- **Semi-synchronous** ($SSYNC$): in this case there is a **global clock** tick reaching all robots simultaneously. At each clock tick every robot is either active or inactive, and only active robots perform their cycle atomically;
- **Asynchronous** ($ASYNC$): there is **no global clock** and robots do not have a common notion of time. Each robot becomes active at unpredictable time instants, and each computation and movement takes a finite but unpredictable amount of time. Finally, only the Looking phase is atomic.

In this sense, a specific problem can be classified as solvable or not depending on the model we're dealing with:

- **Gathering** of 2 robots is solvable in $FSYNC$, but unsolvable in $SSYINC$;
- **Communication** of a coordinate system is solvable in $SSYNC$, but unsolvable in $ASYNC$.

7.2.5 Memory

- **When Sleeping**, a robot forgets everything it has seen during the last cycle;
- **When Looking** again, a robot starts from scratch, with no memory from the past. Thus, every time is the first time.

7.2.6 Why oblivious?

- Robots can **crash** (and recover at a later time);
- Robots may **join** at any time, in any state;
- **Tolerance** to **memory faults**, motorial errors.

7.3 The gathering problem

In this case, initially the robots are in arbitrary distinct **positions**, and in finite time, the goal is to **gather** in the same place.

7.3.1 Gathering in the FSYNC model

Here we have **instantaneous activities**, and all the robots are **active** in every round. Thus, the problem is easily solvable.
An example of solution could be to go towards the **Center-of-Gravity** of the robot group:

$$c = \frac{1}{N}\sum_{i=1}^{N} p_i$$

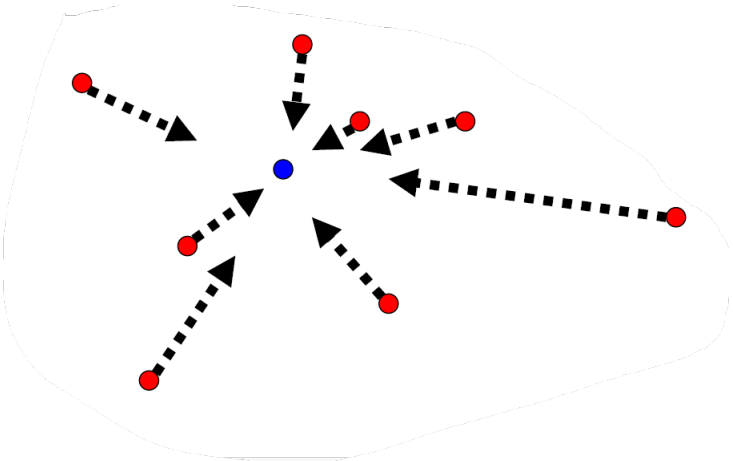


Figure 65: *Gathering in FSYNC.*

Thus, the **strategy** is the following:

1. Once all robots are within distance s_{min} of center of gravity, they **meet** in one round;
2. Until then, in each round, robots get closer to center of gravity by at least s_{min}.

An additional **complication** could be given by the fact that the center of mass could change from one round to the next one.

7.3.2 Gathering in SSYNC/ASYNC

An easy solution to the gathering problem is described as follows.
Given $r_1, .., r_n$, the **Weber point** WP is the point that minimizes the sum of distances to it, i.e.

$$WP = \arg\min_{p \in R^2} \sum_i \text{dist}(p, r_i)$$

The Weber point has the following **properties**:

- It is **unique**;
- It is also the **Weber point** of **other points** on the line $[r_i, WP]$. Thus, WP is **invariant** under robot movements toward it;

Finally, the proposed algorithm for solving the gathering problem is the following:

1. Compute the Weber point WP;
2. Move towards WP.

The main problem is that WP is **not computable**, even for $N = 5$.

Theorem. *For $N = 2$, the gathering problem is **unsolvable** in the SSYNC model.*

Indeed, while in the *FSYNC* model the robots can go the middle point, in the *SSYNC* model an adversary might wake up only one robot each round. In this case we would achieve convergence but not gathering. Some **alternative** ideas are:

- *Each robot goes to other's location?* In this case the adversary will wake up both;
- *One robot goes, one stays?* Adversary will wake up the staying.

Thus, some possible **rules** for robot are:

1. **Go to other robot's place**;
2. **Stay** in place;
3. Go to some other (vacant) **point**.

Then, the correspondent adversarial responses will be:

- If both robots apply **rule 1**: **wake** both;
- **Otherwise**: **wake one robot** that does not apply rule 1.

Randomized gathering of two robots In this case the possible rules for a robot are:

1. **Go to other robot's place**;
2. **Stay** in place.

The algorithm is:

1. **Flip** a fair coin;
2. Apply **rule 1 or 2** accordingly.

Lemma. *In each round, the robots **gather** with probability at least $\frac{1}{2}$.*

Theorem. *For $N = 3, 4, ..,$ the gathering problem is solvable.*

Gathering in SSYNC, $N = 3$ In this case we may face 3 situations:

- The robots are placed in a **line**, so they have to meet in the **middle**;
- Robots are in a **triangle** with an angle with **more** than 120°, so they meet on the robot that is placed in the **vertex** with this angle;
- Robots are in a general **triangle**, they meet at c_e, the **center of equi-angularity**, which is **invariant** under movements towards it.

Figure 66: *3-Gather. First situation.*

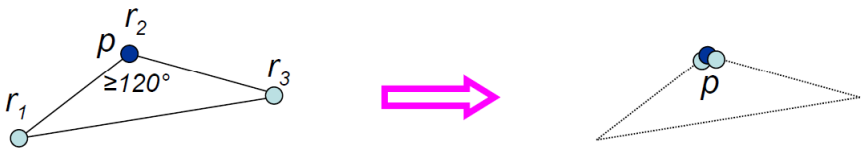


Figure 67: *3-Gather. Second situation.*

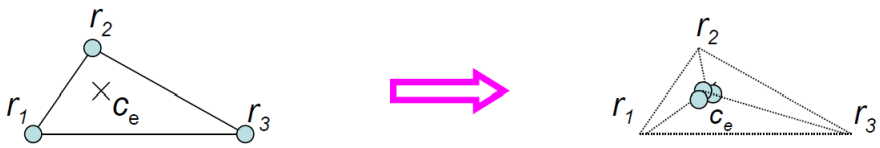


Figure 68: *Gather. Third situation.*

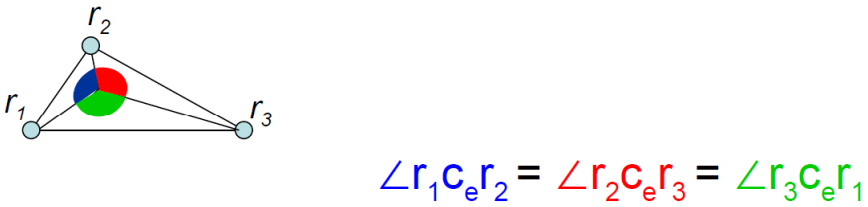


Figure 69: *Properties of the center of equi-angularity. Such properties make this point invariant under the movements towards it, i.e. movements towards it do not change it.*

Gathering in SSYNC, $N \geq 3$ Before analyzing this method, we define a **multiplicity point**. A multiplicity point p^* is a point where two or more robots reside. By assumption:

- **Initially**, there are **no** multiplicity points (namely, each robot is in a distinct location);
- Robots can **detect** multiplicity.

The strategy for solving this problem consists of two stages:

1. **Stage A**: create a single multiplicity point p^*;
2. **Stage B**: move to p^* along "free corridors".

Notice that if two or more multiplicity points occur, the problem becomes **unsolvable** (intuitively, we are not able to break the symmetry, as in the case of $N = 2$). In this sense, multiplicity points can be created accidentally, if:

- The **paths** of two robots intersect;
- The robots **halt** prematurely (e.g., after moving a distance $\geq S_{\min}$).

Stage A: Create a multiplicity point

In order to solve Stage A, a basic tool we can exploit is the **Smallest Enclosing Circle** (**SEC**). For a given point configuration, the SEC:

- Is **unique**;
- Is **computable** in **polynomial time**;
- Is **invariant** while the points on it do not move.

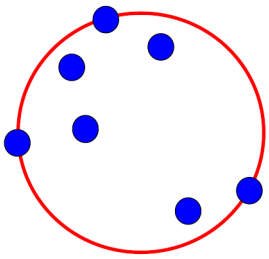


Figure 70: *SEC.*

The following **recursive** procedure *CreateMult* can be used to create a multiplicity point.

1. If $N = 3$, invoke *3-Gather*;
2. Otherwise, calculate the Smallest Enclosing Circle (**SEC**), and let C_{int} be the number of robots inside SEC. Then:
   - If $|C_{\text{int}}| = 0$, go to the **center** of SEC;

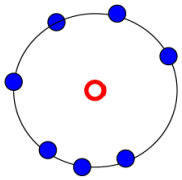


Figure 71: *Case #1.*

   - If $|C_{\text{int}}| = 1$, go to the **internal robot**;

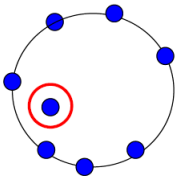


Figure 72: *Case #2.*

   - If $|C_{\text{int}}| = 2$, the two internal robots create a **multiplicity point**. More specifically, we compute the **Voronoi** partition of the robots on SEC, and we continue by case analysis:
     - If the internal robots **do not share a cell**, move to the **center of SEC**;

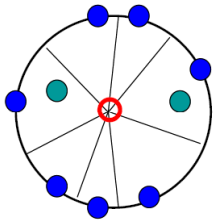


Figure 73: *Case #3.*

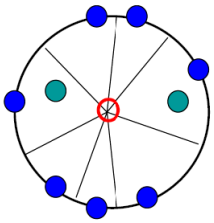


Figure 74: *Case #3a.*

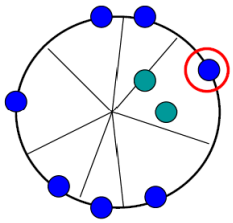


Figure 75: *Case #3b.*

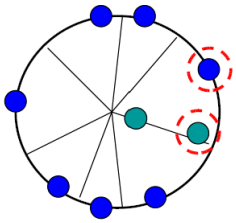


Figure 76: *Case #3c.*

  - If the internal robots **share one cell**, move to the robot defining the **cell**;
  - If the internal robots **share two cells**, then the **inner robot** moves to the **outer** robot, while the **outer robot** moves to one of the **defining robots**.
- If $|C_{\text{int}}| \geq 3$, invoke the procedure **recursively** for internal robots (external robots remain stationary).

Notice that the **trajectories** of robots moving towards p_G **never intersect**, thus the robots will **never** create additional **multiplicity points** on their way to p_G.

Stage B: Moving to the multiplicity point

We define the following procedure *GoToMult* for moving towards the multiplicity point defined at Stage A.

- If you have a **free corridor**, go to p^*;

- **Otherwise** (if other robots block the trajectory), go to counterclockwise $\frac{1}{3}$ angle to closest robot.

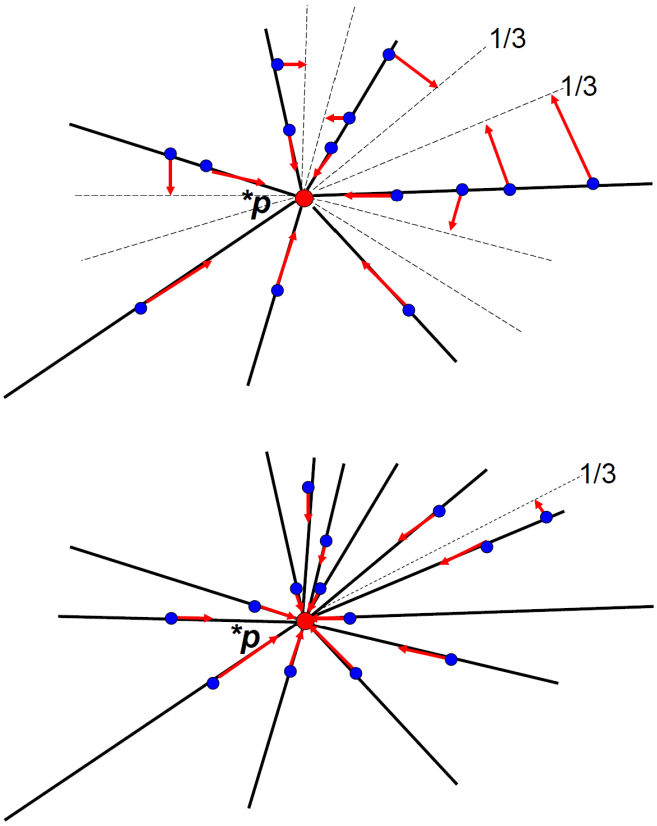


Figure 77: *GoToMult.*

Gathering in ASYNC We recall that in this case we have complete asynchrony:

- Robots operate in **different** and **variable rates**;
- Robots may have **arbitrary wait periods** between cycles.

Notice that for $N = 3, 4$ the simple algorithms for the SSYNC model still work.

7.4 Flocking

Followers recognize Leader, with unlimited visibility and there is no agreement on local axes.

- **Leader** acts **independently** (e.g., human driven);

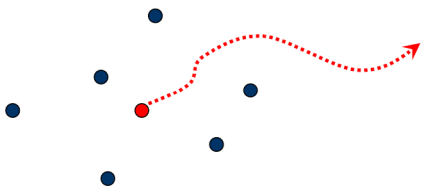


Figure 78: *Flocking: Leader.*

- **Followers** must **converge** to a (commonly known) **pattern** and (approximately) keep it.

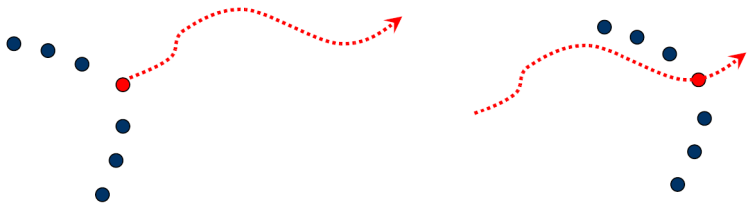


Figure 79: *Flocking: Followers.*

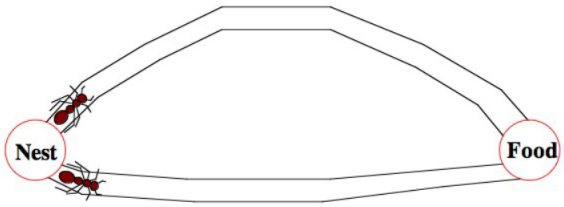
7.5 Ant Robotics

7.5.1 Biological inspiration

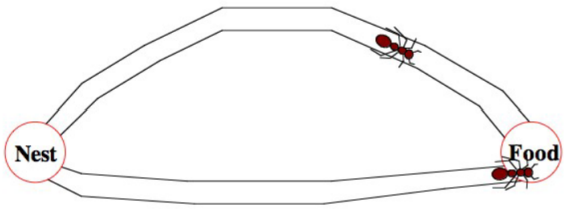
Inspired by foraging behavior of ants, i.e. they find **shortest paths** from **nest** to **food** source. Moreover, ants deposit **pheromones** along traveled path, which is used by other ants to **follow the trail**. This kind of **indirect communication** via the local environment is called **stigmergy**, and it provides **adaptability**, **robustness** and **redundancy**.

7.5.2 Foraging behaviour of ants

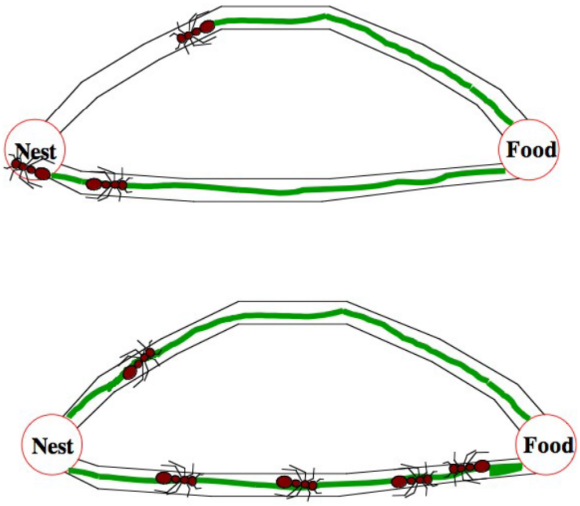
1. Two ants start searching from the nest with equal probability of going on either path;



2. The ant on the shorter path completes the round-trip faster;



3. The density of pheromones on the shorter path is higher (2 ant passes as opposed to 1);
4. The next ant takes the shorter route;
5. Over many iterations, more ants use the path with higher pheromone level, thereby further reinforcing it;
6. After some time, the shorter path is used almost exclusively.



7.5.3 An colony optimization

Simulating this technique leads to the algorithm of Ant Colony Optimization within the area of Swarm Intelligence in AI.