Advanced Algorithms and Programming Methods: Distributed Algorithms

January 7, 2019

BROADCAST: THE FLOODING ALGORITHM

Definition
Consider a distributed computing system where only one entity x knows some important information; this entity would like to share this knowledge with all the other entities in the system. This problem is called *broadcasting*.
Solving this problem means designing a set of rules that, when executed by the entities, will lead to a configuration in which all entities know the information; the solution must work regardless of which entity has the information at the beginning.
According to this, broadcasting requires connectivity restrictions to be solved: in particular, every entity has to be reachable from every other entity.

Time and Message complexity
We will denote by $M(Bcast)$ and $T(Bcast)$ the message and time complexity.
The first message is sent by the initiator and has to reach all the entities.
Given $d(a,b)$ the distance, i.e., the minimal number of edges between a and b, the ideal time depends on the time spent to go from the initiator to any node, thus the *eccentricity* or radius of G.

$$r(initiator) = Max\{d(x,y) : x,y \in V\}$$

Given that any node can be the initiator, we need to compute the diameter of the graph:

$$T(Bcast) = diameter(G) \leq n-1 = \mathcal{O}(n)$$

We consider m as the number of links. We could say that there will be ≤ 2 messages on each link.
Let s be our initiator:

$$M(Bcast) = |N(s)| + \sum_{x \neq s} (|N(x)| - 1) = \sum_{x} |N(x)| - \sum_{x \neq s} 1 = 2m - (n-1)$$

The *flooding* algorithm has optimal time complexity, but the number of messages sent can be reduced! In fact, $M(BCast) \geq m = \Omega(m)$ and $M(Flooding) = 2m - n + 1$.

Assumptions
Before proceeding with the actual code, we need to make some assumptions about the solution we are providing. In particular:

- Unique initiator: the initiator is unique by definition of the problem
- Total reliability & bidirectional link: to simplify the solution
- G is connected: otherwise the problem would be unsolvable for a given initiator K

Code
We have to consider three different states -INITIATOR, SLEEPING, DONE- and two possible events -spontaneous impulse event and receiving of a message.

```
if INITIATOR
        spontaneous impulse event
                send(I) to N(X)
                become (DONE)
        receiving(I)
                do-nothing
if SLEEPING
        receiving(I)
                send(I) to N(x) - {sender}
                become(DONE)
        spontaneous impulse event
                do-nothing
if DONE
        receiving(I)
                do-nothing
        spontaneous impulse event
                do-nothing
```

Notice that if the system is asynchronous, many different executions are possible, depending on the speed at which messages travel.
It is also worth noting that in a distributed environment the entities end their computations at different points in time, hence there is only a concept of *local termination* instead of a *global* one. No entity knows when the entire process terminates!

PROTOCOL SHOUT

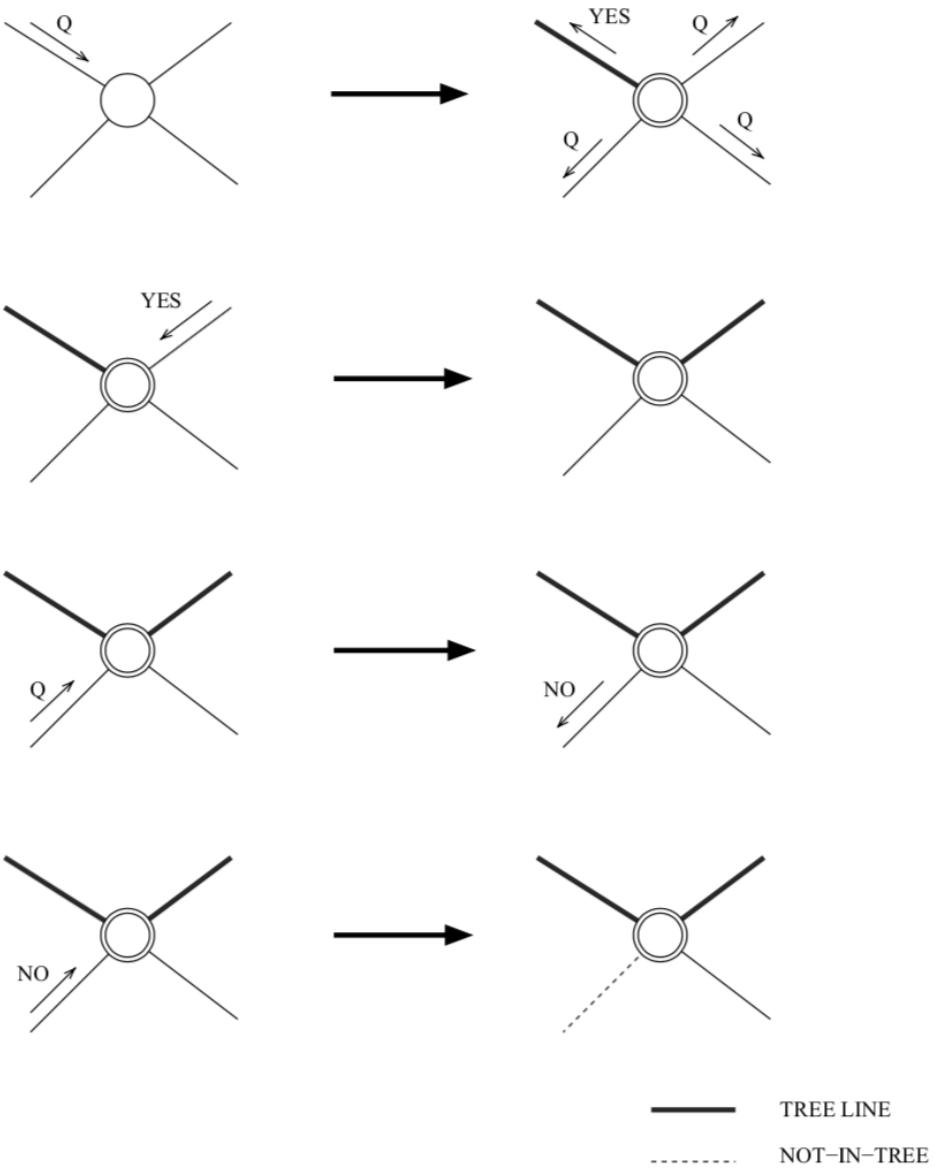
We are assuming that there is a single initiator, there are bidirectional links, there is total reliability and G is connected.

Preliminary: definition of Spanning Tree
A spanning tree T of a graph G = (V, E) is an acyclic subgraph of G such that T' = (V, E') and E' $\subseteq$ E.

Definition
We want to build a spanning tree in a distributed system, but we have to consider that the entities 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 and the fact that, if it sends a message to a neighbor, the message will eventually be received.

Solution *Ask-Your-Neighbors*

- The initiator s will send a message Q = ('Are you my neighbor in the spanning tree?') to all its neighbors
- 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'.
- Each entity terminates when it has received a reply from all neighbors to which it asked the question.



Example of SHOUT

Code: We have four states, INITIATOR, IDLE, ACTIVE, DONE; the terminal state is DONE.

```
if INITIATOR
        spontaneous impulse event
                root = true
                tree-neighbors = {}
                send(Q) to N(x)
                counter = 0
                become ACTIVE
if IDLE
        receiving(Q)
                root = false
                parent = sender
                tree-neighbors = {sender}
                send('Yes') to sender
                counter = 1
                //managing leaves and intermediate node cases
                if counter = |N(x)| then
                        become DONE
                else
                        send (Q) to N(x) - {sender}
                        become ACTIVE
if ACTIVE
        receiving(Q)
                send ('No') to sender
        receiving('Yes')
                tree-neighbors = tree-neighbors ∪ sender
                counter += 1
                if counter = |N(x)|
                        become DONE
        receiving('No')
                counter += 1
                if counter = |N(x)|
                        become DONE
```

Complexity

First of all, notice that *shout* can be assimilated to a combination of *flood* and *reply*. In other words, $M(SHOUT) = 2 \times M(FLOOD)$. To compute the total amount of messages sent in this protocol we have to consider the messages sent by the initiator, plus the other messages Q sent, plus the amount of 'Yes' and 'No' answers.

$$\begin{aligned} M(SHOUT) &= Q + NO + YES \\ &= (n-1) + 2 \times (m-(n-1)) + 2 \times (m-(n-1)) + (n-1) \\ &= 4m - 2n + 2 = 2 \times (2m - n + 1) \end{aligned}$$

SHOUT+

We might wonder that SHOUT could be improved. For example, we might avoid sending all the 'No' messages.
This is the idea behind the SHOUT+ algorithm. In fact, the code is almost the same as SHOUT, but we do not need to send the 'No' message. Of course, this change needs to be managed. In order to do this, we only need to change the ACTIVE state.

```
if ACTIVE
        receiving(Q)
                send ('No') to sender
        receiving('Yes')
                tree-neighbors = tree-neighbors ∪ sender
                counter += 1
                if counter = |N(x)|
                        become DONE
        receiving('No')
                counter += 1
                if counter = |N(x)|
                        become DONE
```

is replaced by

```
if ACTIVE
        receiving(Q)
                counter += 1
                if counter = |N(x)|
                        become (DONE)
        receiving('Yes')
                tree-neighbors = tree-neighbors ∪ sender
                counter += 1
                if counter = |N(x)|
                        become DONE
```

Complexity
The messages are much less. $M(SHOUT+) = 2m$ while $M(SHOUT) = 2(2m - n + 1)$

LEADER ELECTION

Definition
Moving the system from an initial configuration where all entities are in the same state (called *available*), into a final configuration where all entities are in the same state (called *follower*), except one which is in a different state (called *leader*).
We assume a bidirectional graph, connected and no failures.

SATURATION TECHNIQUE

This technique is used to solve many other problems, independently from the number and positions of the initiators (there might be multiple initiators).
The states are *available*, *awake* and *processing*. The algorithm can be divided in three parts:

- Activation phase: started by all initiators, consists in a “wake-up" - all nodes are *activated*
- Saturation phase: started by the leaves, a unique pair of neighbors is identified (*saturated nodes*), the other become *processing*.
- Resolution phase: started by the saturated nodes, it varies depending on the protocol

Code
We have four states AVAILABLE, ACTIVE, PROCESSING, SATURATED. The initial state is AVAILABLE.

```
if AVAILABLE     I haven't been activated yet
        spontaneously
                send('Activate') to N(x)
                neighbors = N(x)
                if |neighbors| = 1    leaf case
                        parent = neighbors
                        send('Saturation') to parent
                        become(PROCESSING)
                else
                        become(ACTIVE)
        receiving('Activate')
                send('Activate') to N(x) - {sender}
                neighbors = N(x)
                if |neighbors| = 1
                        parent = neighbors
                        send('Saturation') to parent
                        become(PROCESSING)
                else
                        become(ACTIVE)
```

```
if ACTIVE
        receiving(M)
        neighbors = neighbors − {sender}
        if |neighbors| = 1
                parent = neighbors
                send('Saturation') to parent
                become(PROCESSING)

if PROCESSING
        receiving(M)
                become(SATURATED)
```

Complexity

$$M(Saturation) = 2n-2+n+n-2 = 4(n-1)$$

But how can this technique be useful for our election problem?

SOLUTION TO THE ELECTION PROBLEM

Let us assume we assign a label $v(x)$ to each node (notice that if nodes are indistinguishable, ranking is unsolvable). Then, it is possible to:

1. Run the saturation technique
2. The two saturated nodes exchange the value and if it is different they can elect as a leader the one with smaller value.
   Observations:
   - If nodes do not have distinct labels, the leader cannot be elected.
   - Each saturated node knows where the other one is, since it knows from which edge it has received the last message.

Each entity x has in input its value $value(x)$. At the end each entity should know whether it is the minimum or not.

Code

The states are AVAILABLE, ACTIVE, PROCESSING, MINIMUM, LARGE. The starting state is AVAILABLE.

```
if AVAILABLE
        spontaneously
                send('Activate') to N(x)
                min = v(x)
                neighbors = N(x)
                if |neigbours| = 1 //leaf
                        M = ('Saturation', min)
                        parent = neighbors
                        send(M) to parent
                        become(PROCESSING)
                else
                        become(ACTIVE)
        receiving('Activate')
                send('Activate') to N(x) - {sender}
                min = v(x)
                neighbors = N(x)
                if |neighbors| = 1      //leaf
                        M = ('Saturation', min)
                        parent = neighbors
                        send(M) to parent
                        become(PROCESSING)
                else
                        become(ACTIVE)
if ACTIVE
        receiving(M)
                min = min{min, M}
                neighbors = neighbors - {sender}
                if |neighbors| = 1 //leaf
                        M = ('Saturation', min)
                        parent = neighbors
                        send(M) to parent
                        become(PROCESSING)
```

```
if PROCESSING
        receiving(M)
                min = MIN{min, M}
                notification = ('Resolution', min)
                send (Notification) to N(X) - parent
                if v(x) = min
                        become MINIMUM
                else
                        become LARGE
```

Complexity
$M(minimum) = 2 \times (n-1) + n + n - 2$ where $2 \times (n-1)$ is the cost for the activation; n the cost for saturation and $n-2$ the cost for notification.$M(minimum) = 4n - 4 = \mathcal{O}(n)$

RANKING PROBLEM IN ARBITRARY NETWORKS

Definition:
Given a graph G(V, E) rank it from the lowest value to the largest.

Solution
In an arbitrary network, this problem could be solved in the following way:

- Find a spanning tree
- Use saturation and the minimum finding to find a starting node
- Do ranking (Centralized or De-centralized)

CENTRALIZED RANKING

This problem could be solved in the following way:

- Build a spanning tree using SHOUT algorithm
- Elect a leader (minimum) using saturation technique
- The leader knows the minimum, it sends in that direction a ranking message.
- Every node knows the minimum in its subtrees, it can then forward the ranking message (ranking, minimum) in the right direction.
- When the node to be ranked receives the message it sends up a notification and update message *(new-minimum)* that will travel up to the leader.

Complexity

$$M(rank\_centr) = 2 \times (n-1) + 2 \times (n-2) + ... + 2(1) = n \times (n-1)$$

Solution
The starter node sends a ranking message of the form *(first, second, rank)* in the direction of the first. The first has the smallest value, the second the second smallest known so far. If no value is indicated (or the value is ∞) it means that the smallest in the corresponding subtree is unknown (at the moment).
The ranked node attempts to send a ranking message to the next node to be ranked. 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.

Complexity
$M(rank\_decentr) = 2\times(n-1)+(n-2)+(n-3)+...+1 = (n-1)n/2+(n-1) = (n-1)(\frac{n}{2}+1)$

Code

- Build a spanning tree using SHOUT
- We elect a leader (e.i. minimum using saturation technique)
- Jump from the min to the second one and so on using messages of the form *(first, second, #rank)*. After having elected the minimum, we know the second smallest element and so we send a message with *(secondValue, ∞ , 2nd)*: this message travels across the spanning tree to find the third element. After the second receives the message, set the value of the subtree as ∞ because there is nothing to rank in that subtree - and so on.

ELECTION PROBLEM

The election problem cannot be generally solved if the entities do not have different identities.
Consider a synchronous system where the entities are unique, have the same state and are anonymous. At each moment, they are doing the same thing, receive the same messages, move to the same states.
In such a system, it is impossible to elect a leader, because the entities are indistinguishable. That is why we need entities with different identities.
Notice that with distinct IDs, minimum finding is an election.

Election in trees
To each node x is associated a distinct identifier v(x).
A simple algorithm:

- Execute the saturation technique
- Choose the saturated node holding the minimum value

Election in rings

A ring is a sparse network (few arcs) such as trees, but it's not symmetric. Every entity has two neighbors.

Each ID is fully circulated in the ring, hence each entity sees all identities.
Assumptions:

- Two versions: unidirectional and bidirectional
- Local orientation
- Distinct identities

All the entities can be the initiators. The bidirectional version assumes that the ring is oriented. The protocol works even without knowing the dimension of the ring and having messages that arrive in a non-FIFO order.
Ideally: each node sends it value; if it receives a message, it forwards it and it keeps track of the minimum value seen so far.

Code
The states are ASLEEP, AWAKE, FOLLOWER, LEADER; the starting state is ASLEEP and the terminal states are FOLLOWER and LEADER.

```
// notice that INITIALIZE and CHECK are "macros" / functions
INITIALIZE
        count = 0
        size = 1
        known = false
        send('Election', id(x), size) to right
        min = id(x)

CHECK
        if count = ringsize
                if min = id(x)
                        become(LEADER)
                else
                        become(FOLLOWER)

if ASLEEP
        spontaneously
                INITIALIZE
                become(AWAKE)
        receiving('Election', value, counter)
                INITIALIZE
                send('Election', value, counter+1) to other
                min = min{min, value}
                count = count + 1
                become(AWAKE)
```

```
if AWAKE
        receiving('Election', value, counter)
                if value ≠ id(x)
                        send('Election', value, counter+1) to other
                        min = min{min, value}
                        count+=1
                        if known
                                CHECK
                else
                        ringsize = counter
                        known = true
                        CHECK
```

Complexity
Each entity crosses each link, so the complexity is $\mathcal{O}(n)^2$; the size of each message is $log(id)$.
$M(Alltheway) = \mathcal{O}(n)^2$ messages
$T(Alltheway) \leq 2n-1 = \mathcal{O}(n)$

Observations:

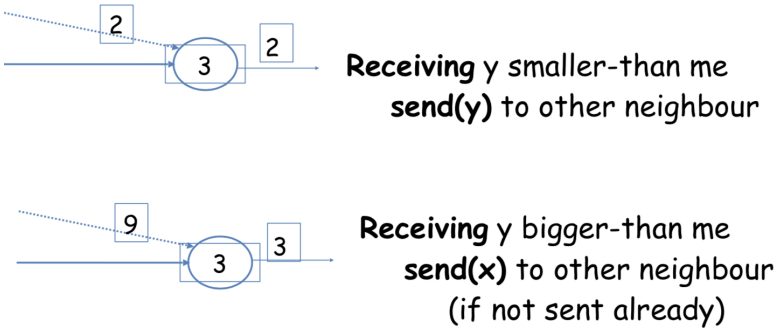
- The algorithm also solves the data collection problem
- It works for both unidirectional and bidirectional cases

ANOTHER SOLUTION: ASFAR (AS IT CAN)

Idea
It is not necessary to send and receive messages with larger IDs than the IDs the current node has already seen.

Assumptions

- Unidirectional/bidirectional ring
- Different id's
- Local oriented



AsFar example

Code

The states are ASLEEP, AWAKE, FOLLOWER, LEADER; the starting state is ASLEEP, the terminal states are FOLLOWER and LEADER.

Unidirectional version

```
if ASLEEP
        spontaneously
                send('Election', id(x)) to right
                min = id(x)
                become(AWAKE)
        receiving('Election', value)
                send('Election', id(x)) to right
                min = id(x)
                if value < min
                        send('Election', value) to right
                        min = value
                become(AWAKE)
if AWAKE
        receiving('Election', value)
                if value < min
                        send('Election', value) to right
                        min = value
                else
                        if value = min
                                NOTIFY
        receiving(NOTIFY)
                send(NOTIFY) to other
                become(FOLLOWER)
NOTIFY
        send(NOTIFY) to right
        become(LEADER)
```

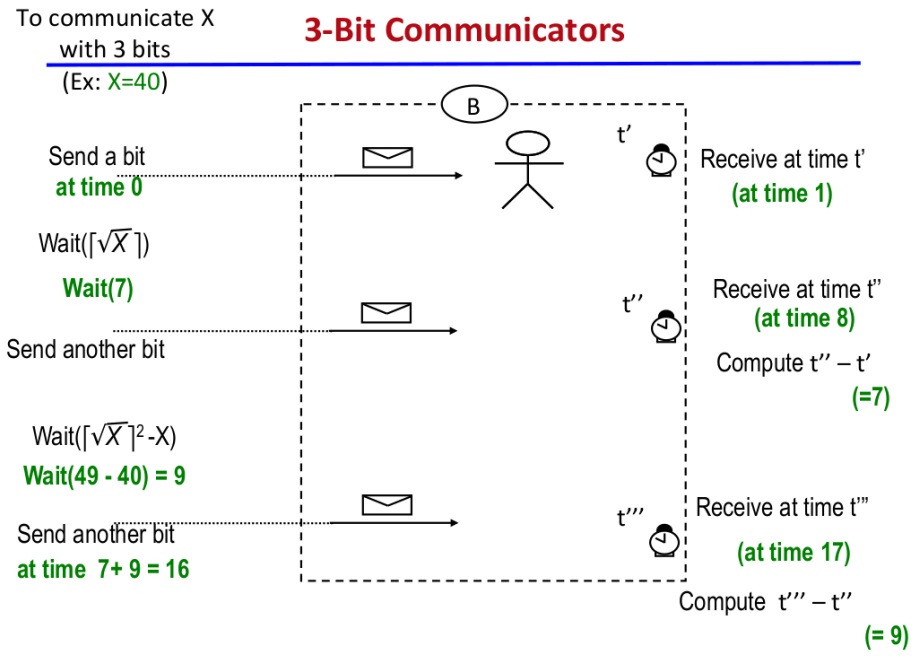
SYNCHRONOUS SYSTEMS

We consider as synchronous system all those systems in which all local clocks are incremented by one unit simultaneously; in other words all local clocks 'tick' in the same moment. Notice that this assumption does not mean that all the clocks have the same value, but just that their values are incremented at the same time.
The first part of this section is about calculating how much time will it take to send a message in a synchronous system.

2-BIT COMMUNICATORS

You want to send the number X. Send a bit, wait X seconds, send another bit. The receiver just needs to compute the difference between the times in which he receives the two messages: that is number X.

3-BIT COMMUNICATORS



3-bit communicators

Time complexity
$\mathcal{O}(n)^2$.

Proof
We send a number K which is the smallest integer that satisfies the condition:

$$K \in \mathbb{N} : (K-1)^2 < X \wedge K^2 >= X$$

This means that $|K^2 - X| < |(K+1)^2 - K^2| = 2K - 1$.
Therefore we have that the waiting time $C : K^2 - C = X$ is bounded, in particular $0 \leq C \leq 2K$.
The time complexity is, therefore, $\mathcal{O}(K + 2K) = \mathcal{O}(K) = \mathcal{O}(\sqrt{X})$.

K-BIT COMMUNICATORS

In general, for a k-bit communicator, we have that we use k bits to send the message and the time complexity is $\mathcal{O}(k \cdot X^{\frac{1}{k}})$.

MIN-FINDING AND ELECTION

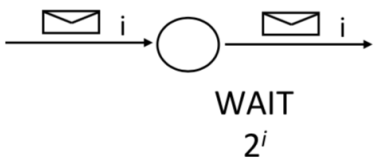
Introduction
Messages travel at different speed. In such a configuration, knowledge of n is not necessary.
We will focus on the synchronous, unidirectional version.
There are two ways of eliminating IDs:

1. similar to AsFar: large IDs are stopped by smaller IDs
2. small IDs travel faster, so they catch up with larger IDs and eliminate them

General idea
It is important to recall that each message travels at a speed which depends on the identity it contains. Speed is assumed to be the same for every message (and unitary). We can change it by introducing appropriate delays.
When a node receives a message containing i, it waits $f(i)$ ticks. 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.

EXPONENTIAL CASE

An example could be done with $f(i) = 2^i$. When a node receives a message containing i, it waits 2^i ticks.



Graphical example of 2^i delay

Complexity
In time $2^i \cdot n + n$ the smallest ID i traverses the ring. Let the second smallest be $i+1$ (waiting time 2^{i+1}). While the smallest ID goes around, the second smallest has the time to traverse $(2^i \cdot n + n)/2^{i+1} \approx n/2$ links.
We can generalize that by thinking about the j^{th} ID: in time $2^i \cdot n + n$, it traverses $(n/2^j)$ links.

The smallest identity *i* is the quickest to go around the ring

Messages:	*n*	O(*n*)
Time:	$2^i n + n$ units	O($2^i n$)

The second smallest id: *i* + 1

in time: $2^i n + n$ has traversed $(2^i n + n)/ 2^{i+1}$ --- $(n / 2)$ links

The third smallest id: *i* + 2
in time: $2^i n + n$ has traversed $(2^i n + n)/ 2^{i+2}$ --- $(n / 4)$ links
...

The j^{th} id
in time: $2^i n + n$ has traversed $2^i n+n) / 2^{i+j}$ --- $(n /2^j)$ links

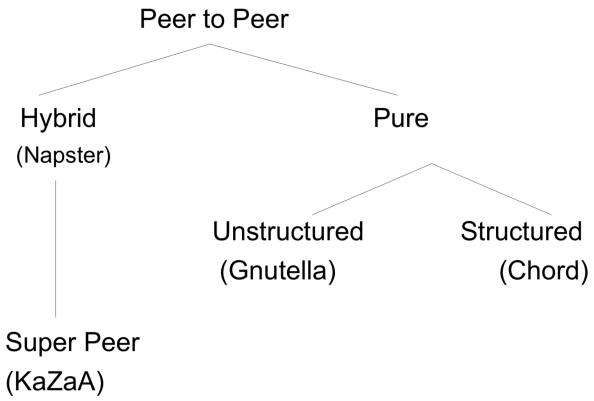
Considering all of the above, we get

$$totalMessages = \sum_{j=1}^{n-1} \frac{n}{2^j} = n \sum_{j=1}^{n-1} 1/2^j = \mathcal{O}(n)$$

Setting i as the smallest ID and Id as the biggest ID we obtain a time complexity of $\mathcal{O}(2^i n)$ and a total message complexity (in bits) of $\mathcal{O}(n \cdot \log(Id))$.

PEER-TO-PEER

ARCHITECTURE

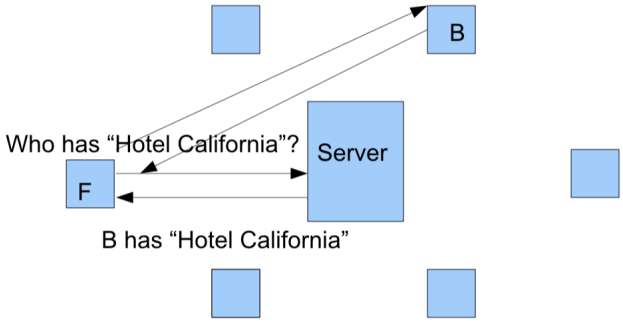


p2p architecture

SEARCH

HYBRID P2P

Napster presents a centralized lookup. 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 is then exchanged among peers.

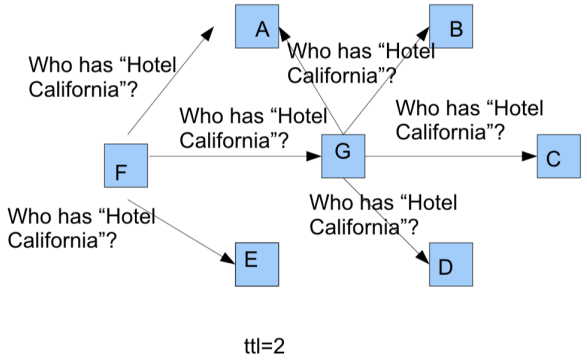


Search in hybrid P2P systems

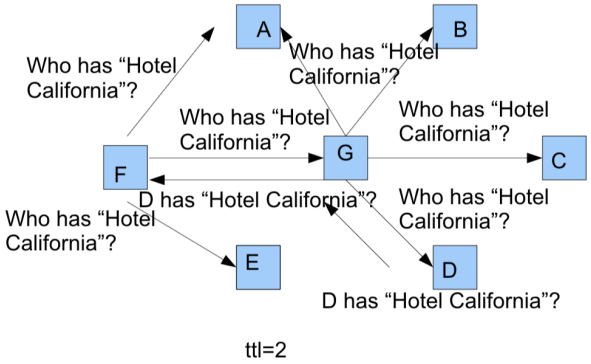
UNSTRUCTURED PURE P2P

Gnutella is a pure P2P without a central server. Peers have an initial set of addresses known for the first connection; peers are equally treated, no matter the bandwidth they have and how many files they share. Each peer provides both the files and sends/replies to routing requests. Each peer is both client and server.

Each peer knows a subset of neighboring 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 peer to the P2P network.
When the search starts, requests from a peer R are sent from neighbor to neighbor (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 (PONG) to R [R's ID is sent together with the request]. Afterwards, R will directly contact P and will download the resource. **Problem:** hard to control/regulate



PING with Gnutella



PONG with Gnutella

ITERATIVE DEEPENING

Gnutella's search method has some limits:

1. R might receive redundant results or nothing
2. the TTL needs to be tuned: a too-low one will result in a resource not found, whilst a too-high one will flood the network with requests (and possibly answers). It is also possible that the too many requests turn into a DDoS!

To address these problems, unstructured pure P2P systems such as Gnutella use a technique called *iterative deepening*.

1. The system is flooded with a limited TTL
2. If the resource is not found, we start with a bigger TTL (the sequence of TTL's is predefined)
3. We repeat up to when the resource is found or when a boundary TTL is found
   Many variations: blind, informed...

SUPER PEER (KAZAA)

Combines characteristics of pure and a hybrid P2P systems.

- There are different servers; the super peers supervise the subnetworks (subtrees)
- The super peers execute the queries for the leaves (i.e. they work as servers)
- The super peers also operate as normal peers
- The peers exchange information directly
- Combine the characteristics of hybrid and pure systems

Search

We need to consider the following questions:

- Good numbers of leaves for each super peer?
- How should super peers connect together?
- The system is made more reliable k redundant

BIT TORRENT

Many users can simultaneously download the same file without delaying too much each other. Files are not downloaded from the same server but they are divided into different portions. Each peer that makes a request offers already downloaded portions to the other peers, thus contributing to the downloading of the other peers.
We concentrate on the efficient fetching and not on the search. There is a single publisher and many downloaders. The same file is distributed among many peers.

INFORMATION STORAGE

The main questions are: where to store information in a P2P system? How to find it?
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

There could be many approaches to store our information.

- Centralized approach: we send a request to the server. We use $\mathcal{O}(N)$ memory on the server, search $\mathcal{O}(1)$ step to reach the server
- Decentralized approach: we send a request to all our neighbors (different optimizations, i.e. TTL). We use $\mathcal{O}(1)$ memory and we use $\mathcal{O}(n^2)$ steps for the search.

They offer a decent memory/step tradeoff. We use $\mathcal{O}(\log(N))$ steps to find the information and $\mathcal{O}(\log(N))$ entries in the routing table of each node.
Technique:

- Adaptability: it is simple to insert (adding the information) and remove nodes (reassignment to the neighboring nodes)
- Balances information on the nodes (makes routing efficient)

Peers and data are mapped in the same address space through hash functions. For nodes we use the hash of their IP; for data we use the hash of the content (title, ...).

CHORD

Data distribution
Data are distributed among peers using a precise algorithm. Data are replicated to improve availability.

Assignment

- Each peer has an ID (hash of the IP)
- Each resource has a key (hash of the title, ...)
- We use long keys to avoid collisions
- A single peer stores resources with keys similar to the one of the peer itself (same logical addressing space)
- Given a key of a resource the peer sends the request to the peer with the most similar key

Using Chord we have to use a consistent hashing:

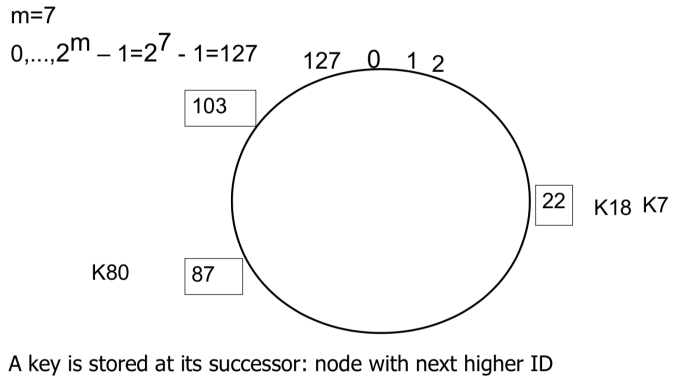
- To assign the keys to the peers
- There should be load balancing so that with very high probability each peer receives the same number of resources/keys
- We ask for key updates when a peer connects/disconnects from the network
- The insertion of the N-th key with high probability will require the movement of 1/N other keys

Advantages

- We maintain consistent 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 $\mathcal{O}(\log(N))$ to reach the destination
- A peer requires a table of $\mathcal{O}(\log(N))$ 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

Idea
The logical space is a circular ring 0,..., $2^m - 1 \pmod{2^m}$. A resource with key K is stored in the

node which is the closest successor from K, that we call *successor(K)*.



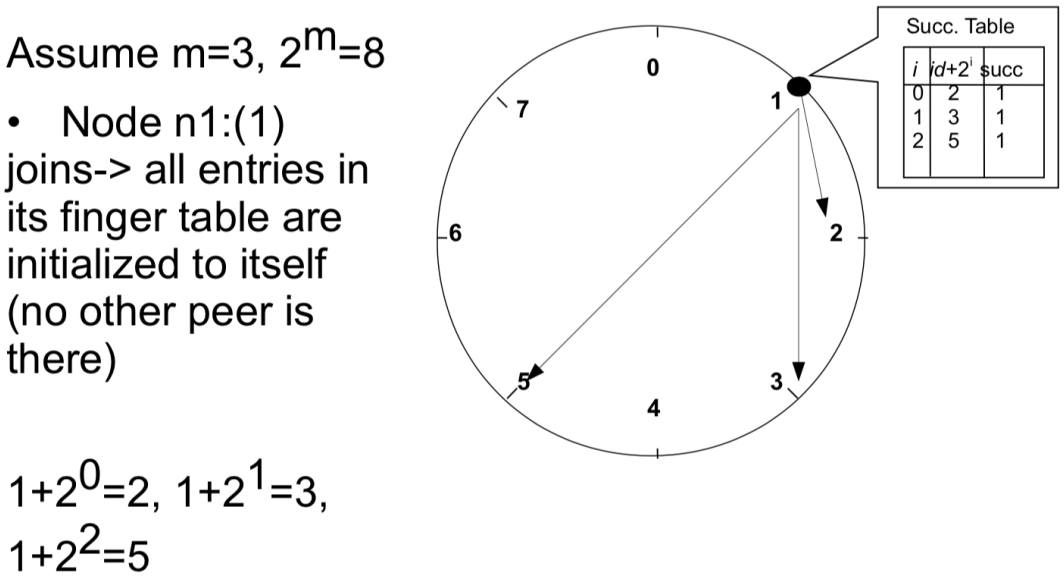
Example of how Chord assigns IDs

Successor: the successor of peer 22 is 87, of 87 is 103, of 103 is 22.
Routing Algorithm #1: each node stores only its successor. If the resource is not on this node, then the query is sent to the successor.
It requires $\mathcal{O}(1)$ of memory, $\mathcal{O}(N)$ for search: in the worst case we have to visit the whole ring. The failure of a node blocks the procedure.
This algorithm, in the worst case, requires too many operations. To solve this problem, let's 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. Also, this protocol works assuming that the value of the next successor is correctly maintained/updated.

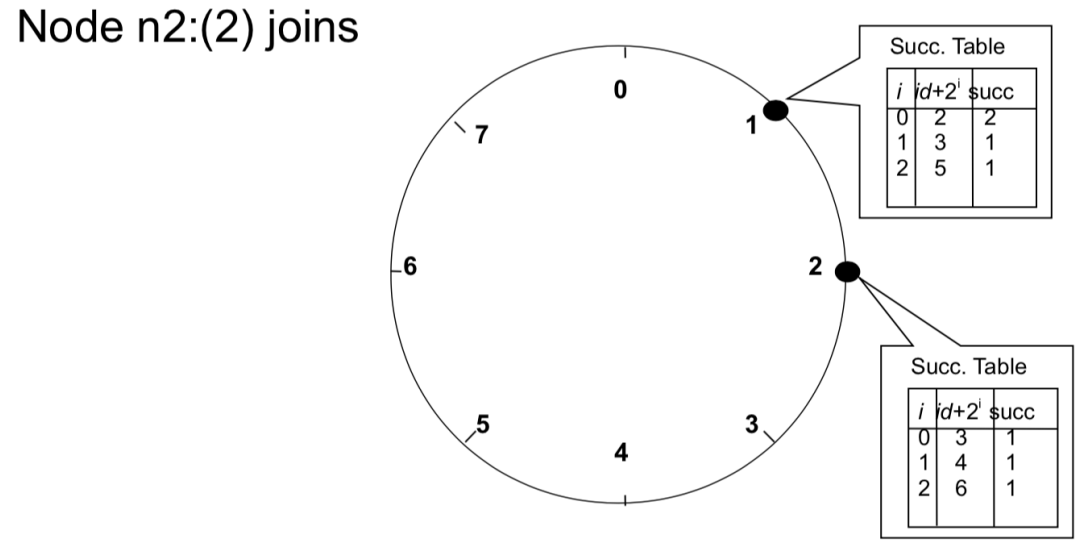
Routing Algorithm #2: This solution is the opposite of #1: each node stores N successors. If a resource K is not on a node, the node looks in the successor node. This requires $\mathcal{O}(N)$ memory and $\mathcal{O}(1)$ for the search.

Routing Algorithm #3: The best solution consists in storing m values in each node. The search of K is sent to the furthest known predecessor of K. We store more values of the close nodes and less of more distant nodes, thus routing is more precise close to a node. This approach requires $\mathcal{O}(\log(N))$ of memory and $\mathcal{O}(\log(N))$ for the search.
Each node maintains a *finger table* (the i-th entry of the finger table of x is the first node that succeeds or equals $(x+2^i) mod 2^m$) and the predecessor node. An item identified by ID is stored on the successor node of ID.

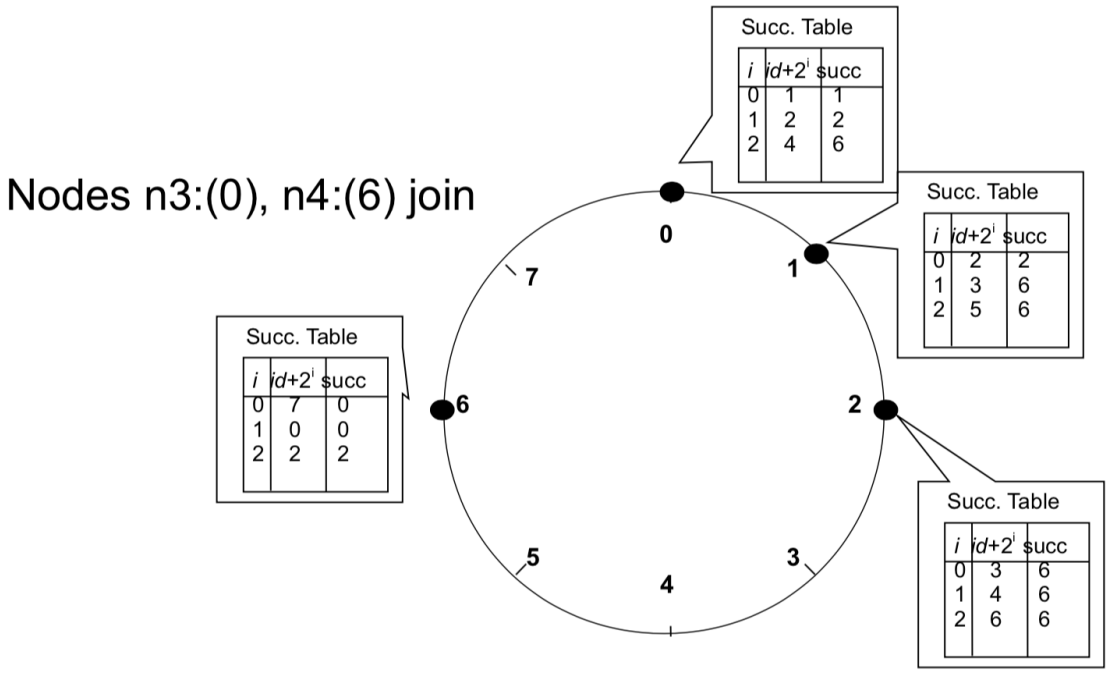
Here below it will be provided some examples of how this algorithm works.



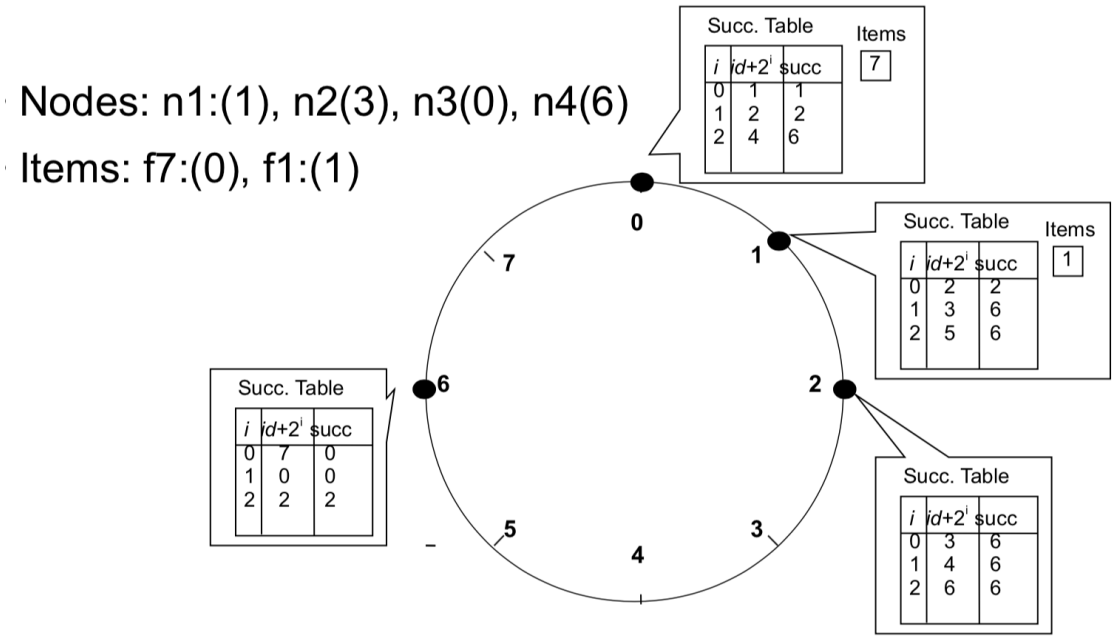
First assignment with Chord



Second node joins

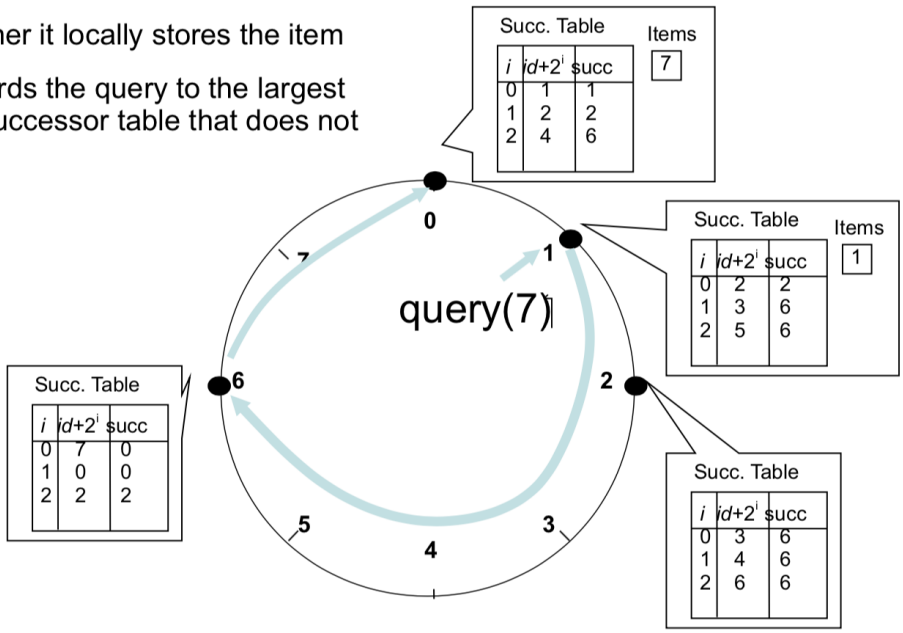


Third and forth joins



Item assignment

- 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*



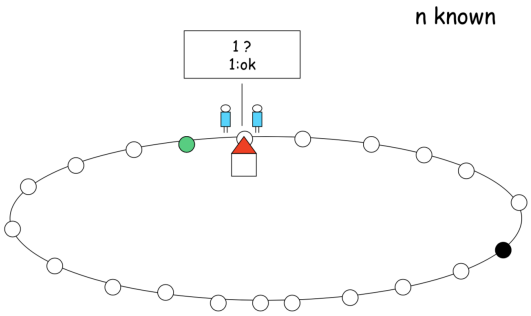
Routing after having received a query

MOBILE AGENTS AND THE BLACK HOLE PROBLEM

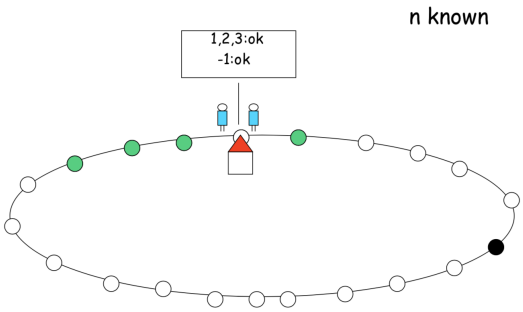
Definition: Consider that we have some mobile agents that can travel across the graph and could communicate together with a whiteboard and they also have some storage, the problem is to find a malicious node (black hole) that makes disappear our mobile agents. Each agent knows the number of nodes.

CAUTIOUS WALK

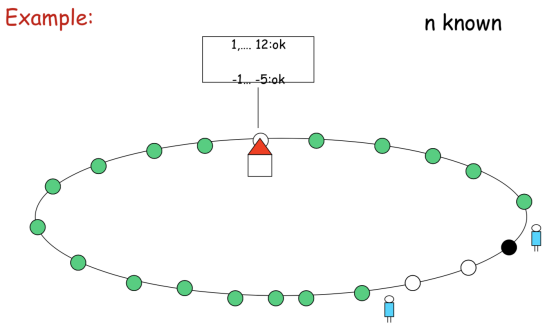
We can consider our graph as a set of ports and in every port there is a traffic light. If the next port has not been controlled, the traffic light is red, going to the next node we activate it (yellow light), and when we come back from that node, that edge (and the next node) is safe (not the black hole).
What about if the configuration of the graph is a ring?
We need two agents!
The agents, starting from the home base, must explore using cautious walk. The agents must explore disjoint areas otherwise they could both disappear. So they finally explore step by step all the ring until one of the two agents disappear of they finish the graph and the last node that has not been marked is the black hole.



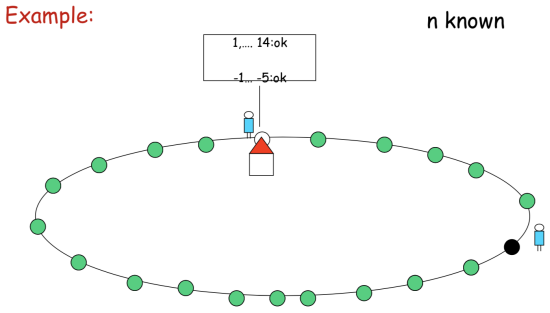
First step of the left agent



First step of the right agent



Black hole for the right agent



Left agent complete the ring and knows where is the black hole (for the whiteboard)

But what if also the intruder is mobile?

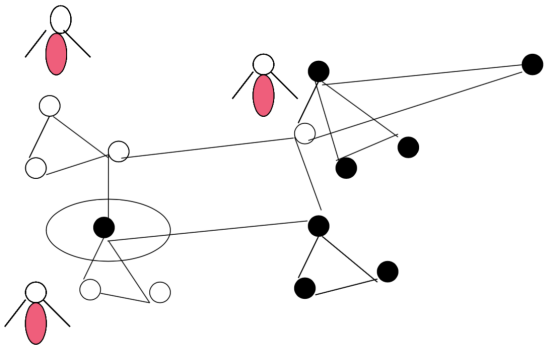
THE INTRUDER CAPTURE PROBLEM

Let's put some rules, the intruder:

- can moves from a node to a neighbouring one
- can moves arbitrarily fast
- cannot cross a node guarded by an agent
- is invisible to the agents
- can permanently see the position of the other agents

Initially the agents are located at the homebase and form a team while in the end the agents capture (surround) the intruder.
The intruder capture problem is equivalent to the decontamination problem.
As said 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 in it. At the end the whole network must be clean.
Contamination rule: a node is contaminated if it is not protected by an agent and at least one of its neighbors is contaminated (remember that the intruder is fast).

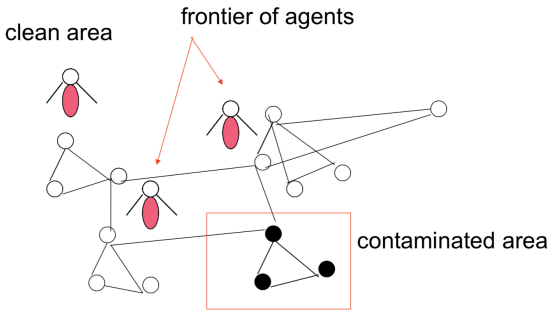


Contaminated node

Decontamination process: As said several times, 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 or an edge when it enters (or traverse the edge) in it, a node should not be recontaminated. A node becomes contaminated if it is not protected by an agent and at least one of its neighbors is contaminated.

CONTIGUOUS MONOTONE STRATEGIES

We consider the situration where agent move only to neighbouring nodes (contiguous) and no recontamination can occur(monotone).



Contaminated node

Complexity issues:

- Number of agents

- Number of moves
- Time (sync or async)
- Memory (agents, nodes(whiteboard))

Strategies are not necessarily connected and monotone (i.e. agents can jump. We have to minimize the number of searchers).
There are two model of visibility, one with visibility of the other neighbors (clean, contaminated, guarded) and the other in which agents have only local knowledge.

DECONTAMINATING A MESH

We have to be in this situation:

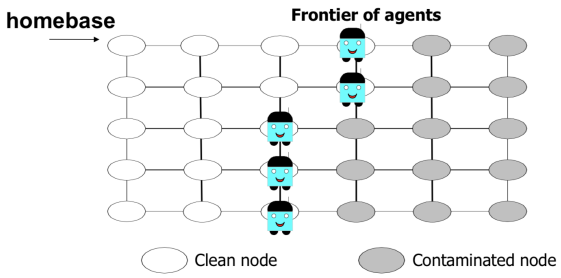
- Asynchronous system
- Node + agent storafe $O(\log(n))$ bits
- Models (visibility, local)

We consider a m x n mesh topology. We have two possible strategies:

- With synchronizer: Searching agents do not have visibility power. The synchronizer is an agent that coordinates the moves.
- Agents with visibility: Visibility power means agent can see their neighbouring nodes, agents can move independently, the synchronizer is not required.

The main idea behind the strategy is to:

- Start from the homebase
- Contiguously clean the contaminated network by maintaining a vertical barrier of agents (to avoid recomtamination, works asynchronously)
- Move one column at the time



Contaminated node

We can perform the search of a node or of an edge moving the barrier. (see slides)