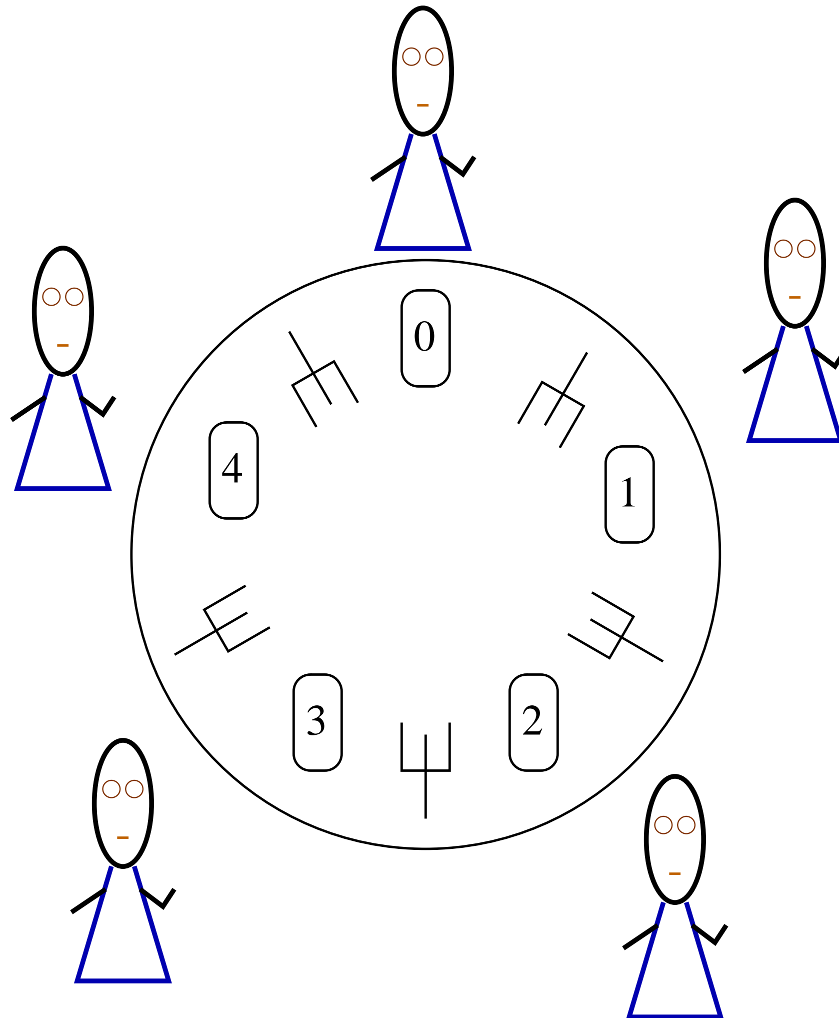
Once upon a time ... [Dij71]

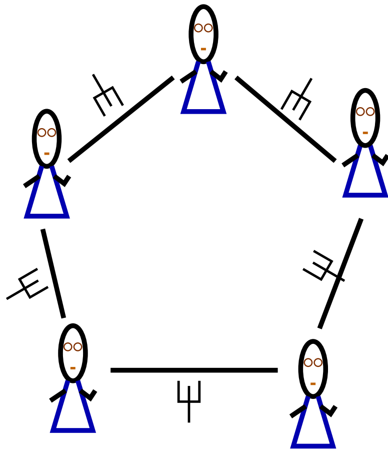


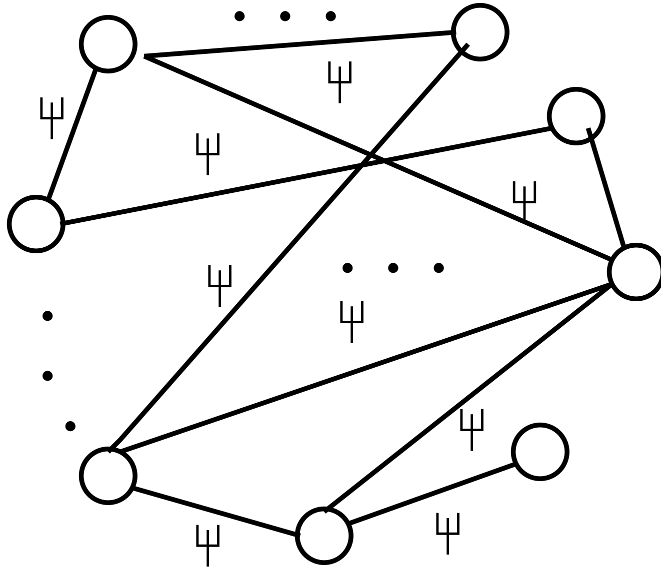
No deterministic symmetric dining solution

[RL81] Probabilistic symmetric solution

Traditional non-symmetric: Left-Right solution

Generalized dining philosophers [Lyn81]





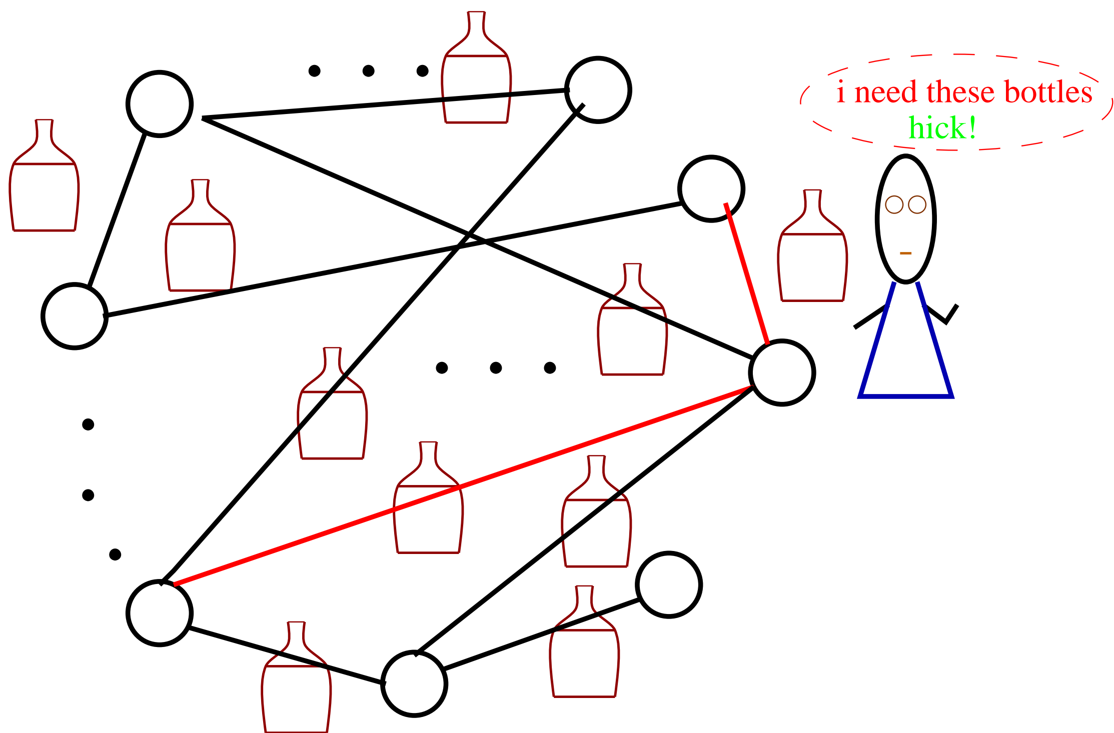
Conflict graph, Δ = degree

- Each philosopher has an arbitrary number of neighbours
- $\exists$ one fork between every pair of neighbours
- Each philosopher needs all adjacent forks to eat
- Requirements:

  Exclusion: Disallow neighbours to eat simultaneously

  + no deadlock, no starvation

Drinking philosophers [CM84]

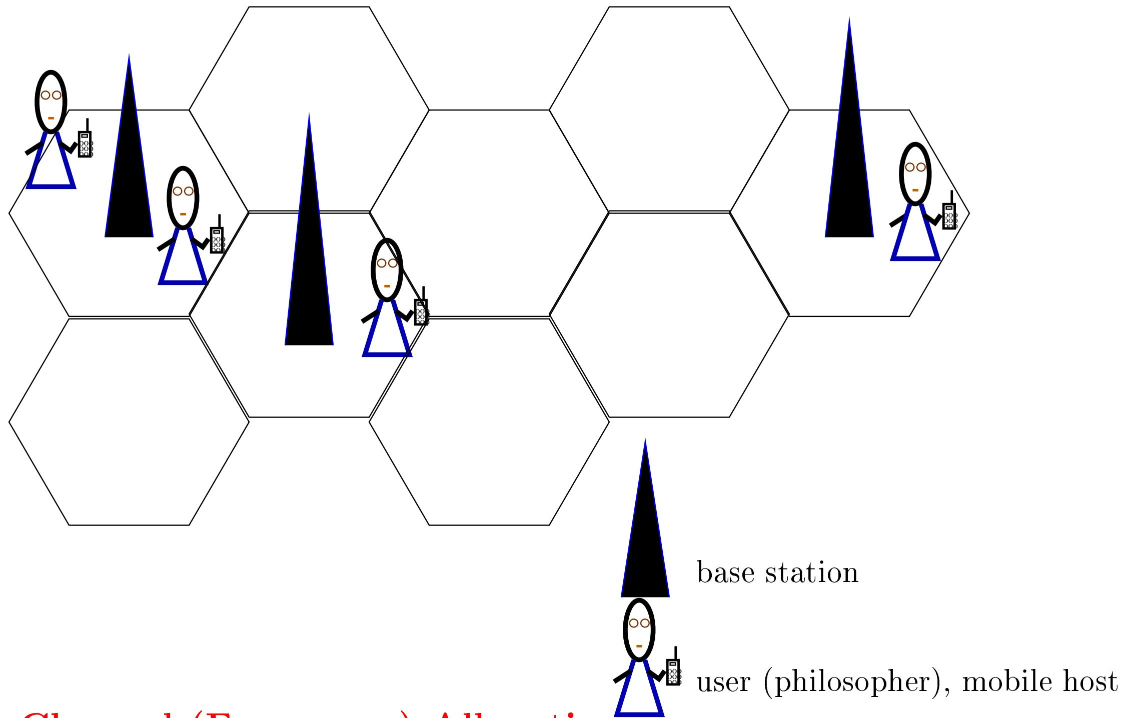


- ∃ one bottle between each pair of neighbours
- Each thirsty philosopher needs a (prespecified) subset (may differ each time) of its incident bottles to drink
- Requirements:

Exclusion: Allow neighbours to drink simultaneously, provided that they need to drink from different (prespecified) bottles

+ no deadlock, no starvation

Mobile Philosophers [GPT96]



Channel (Frequency) Allocation

- **Frequency spectrum same in each cell**
- **Each philosopher needs some (arbitrary, not prespecified) subset of frequencies to communicate**
- **Requirements:**
  Exclusion: Allow choice of frequencies so as to make it possible for neighbours to operate simultaneously
  + no deadlock, no starvation
- **Request satisfiability (Bandwidth Utilisation)**

General Requirements

Exclusion

Different constraints and flexibility for each version

No deadlock

No starvation

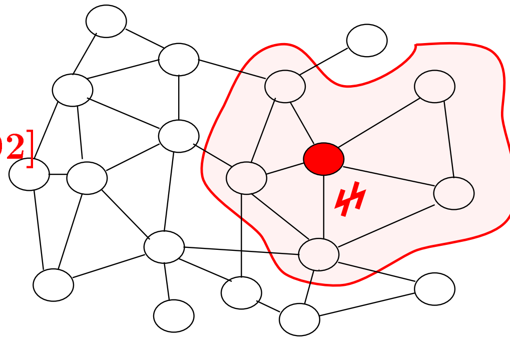
Evaluation Criteria

Time and communication complexity (to satisfy a request)

(units depend on the communication system)

Fault tolerance

Failure locality [CS92]



Request Satisfiability (for mobile philosophers)

Generalized Left-Right Dining [Lyn81]

Idea:

Color resources (edge-color conflict graph), C colors

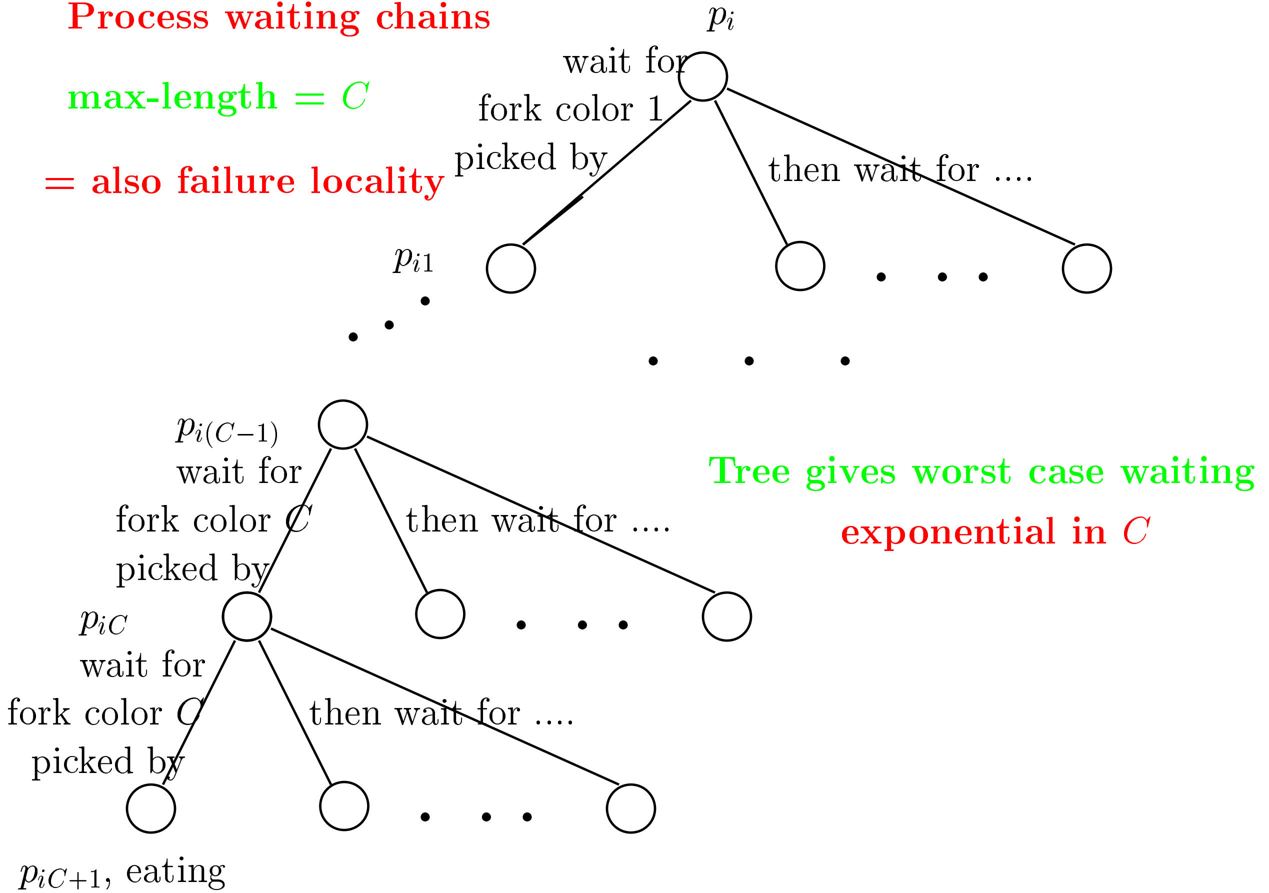
p_i picks forks in increasing color order (using mutex for each)

color order guarantees no deadlock, no starvation

Process waiting chains

max-length = C

= also failure locality



Tree gives worst case waiting exponential in C

Restricting the waiting chain length [SP88]

Idea:

again picks forks in increasing color order (using mutex for each)

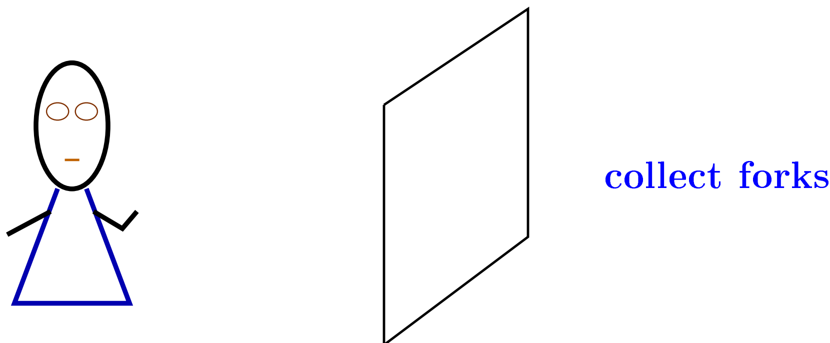
but now:

if p_i picked color 1, 2, . . . $x/2$, . . . need to wait for color x

backtrack (release forks back to $x/2$)

max-waiting-chain-length = $\log C$ (= also failure locality)

in addition: synchronization doorway



collect forks

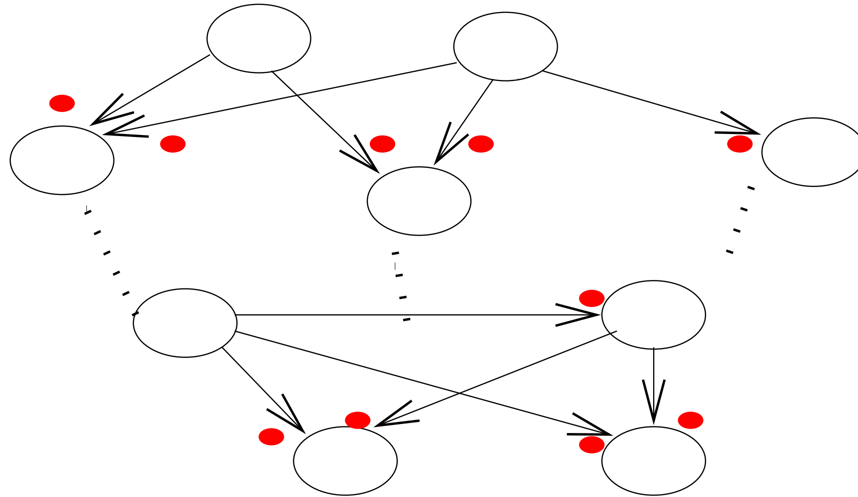
wait for permission by each of the neighbours to cross

Time, communication complexity = $O(\Delta^{\log C})$

Why pick forks 1-by-1? Another approach [CM84]

Idea:

- **Initial Acyclic Orientation**
  using initial labeling (e.g. node coloring) and priviledge tokens
- **Sink-nodes can eat; then**
- **Reverse incident-edge-direction (send priviledges)**



Result:

dynamic priority resolution scheme

can e.g. resolve possible conflicts in the drinking solution

Waiting chain length = O(n) (= failure locality also)

(more on acyclic orientations in [AST94], [BG94])

Conflict -> Precedence Graph

Undirected GRAPH, in which edges represent shared resources between processes we call this graph CONFLICT GRAPH.

The algorithm by Chandy and Misra resolves conflicts by defining for every possible conflict a *precedence* relation:

- When two processes compete for a resource the one with higher precedence may access the resource first.
- In order to receive a solution which is fair these precedences will have to change dynamically.

The directed graph graph that changes dynamically is called *precedence graph*.

For each resource an edge of the precedence graph is directed from processes with lower precedence to processes with higher precedence.

The precedences of the graph are chosen such that it is always possible to distinguish at least one process from all other processes i.e. this process can enter its critical section. (NO DEADLOCK)

This is ensured by the existence of at least one process which has higher precedence for all its shared resources. A process with this property is called *sink*.

Its existence is guaranteed when the precedence graph is always acyclic.

By changing directions of edges it is possible to change the precedences dynamically.

This must happen in a way that the precedence graph stays acyclic, so *progress*, *fairness* and *mutual exclusion* is guaranteed.

Starting with a DAG

- The graph is initialised acyclic for example by a node-colouring algorithm.
- The graph can remain acyclic if after use of the critical section a process reverse all adjacent precedences in one step.
- Need a mechanism to keep the sense of direction:

The mechanism

Forks which have the property to be either *clean* or *dirty*.

- A fork will be cleanedbefore it is send to a neighbour process.
- A clean fork will become dirty when the holder of the resource enters the critical section.
- After use it remains DIRTY until it is sent to a neighbour process.

The dynamic DAG

- The respective precedence graph H can be defined in the following way:

- For all pairs of processes *p* and *q* which share a common resource, <*p*,*q*> one of the following statements is true:

1. *p* holds the *fork* for the resource and the *fork* is CLEAN

2. *q* holds the *fork* for the resource and the *fork* is DIRTY

3. the *fork* for the resource is in transit from *q* to *p*

Requesting Forks

The request of forks is realized by *request tokens*.

For each fork there exist one request token such that only the holder of the request token can request a fork.

A *hungry* process requests a fork by sending the *request* TOKEN to the owner of the desired *fork*.

A process is not interested in accessing its resources when it holds a *request* TOKEN but not a *fork*.

The algorithm

The algorithm is initialised by an acyclic precedence graph H and all processes with lower precedence own dirty forks while processes with higher precedence own request tokens.

All processes are *thinking* i.e they are not interested in their resources.

A process which becomes *hungry* will send all its *request* TOKEN to neighbour processes and wait until it received all *forks*.

- A process which received all forks will change its state to *eating*.
- A process which leaves the CRITICAL SECTION changes the state of all its *forks* to DIRTY. Then for all held *request* TOKEN the respective *fork* is sent to neighbour processes.

The above steps assume following rules:

Receiving a *request* TOKEN for fork f:

1. If processors state is different from *eating* and f is DIRTY then f will be sent to the requesting processor.

2. If processors state was also *hungry* then the *request* TOKEN will also be sent back.

Receiving a fork f: The state of f will be set to clean.

Correctness

Mutual Exclusion:

Proof. The precedence graph H is acyclic. □

No Starvation

Proof. Let the depth in H of any process p be defined as the maximum number of edges along a path from p to another process without predecessor. The proof will show by induction that a process of depth k will eventually eat if predecessors at depth *k-1* can EAT. □

Complexities

Communication Complexity: *O(degree)*

Proof. A process sends at most one *request* TOKEN to each neighbour and receives from each neighbour at most one *fork*. □

Tine Complexity: *O(n)*