Advanced Distributed Algorithms and Data Structures

Chapter 2: Foundations

Christian Scheideler
Institut für Informatik
Universität Paderborn

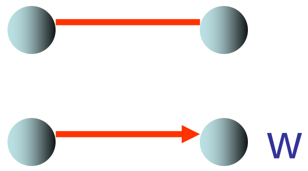
Overview

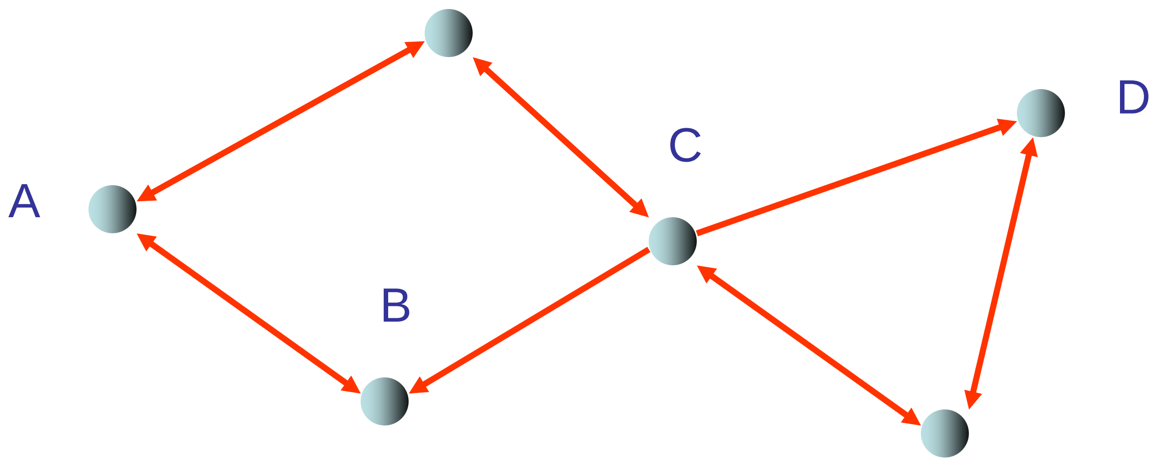
- Graph theory
- Classical graph families
- Fundamental graph parameters
- Process Model
- Pseudo-Code
- Synchronization

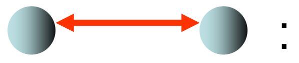
Graph Theory

Definition 2.1: A graph $G=(V,E)$ consists of a node set V and an edge set E.

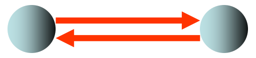
- G undirected: $E \subseteq \{ \{v,w\} \mid v,w \in V\}$
- G directed: $E \subseteq \{ (v,w) \mid v,w \in V\}$







short form of



Graph Theory

Definition 2.2: Let $G=(V,E)$ be a graph.

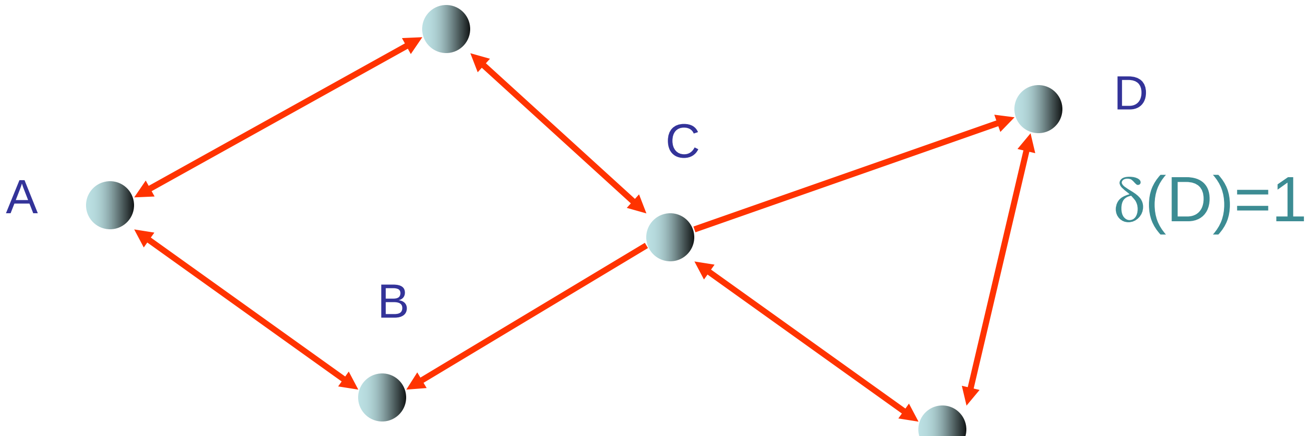
- G undirected: degree of $v \in V$:

  $\delta(v)=|\{ w \in V \mid \{v,w\} \in E\}|$

- G directed: degree of $v \in V$:

  $\delta(v)=|\{ w \in V \mid (v,w) \in E\}|$

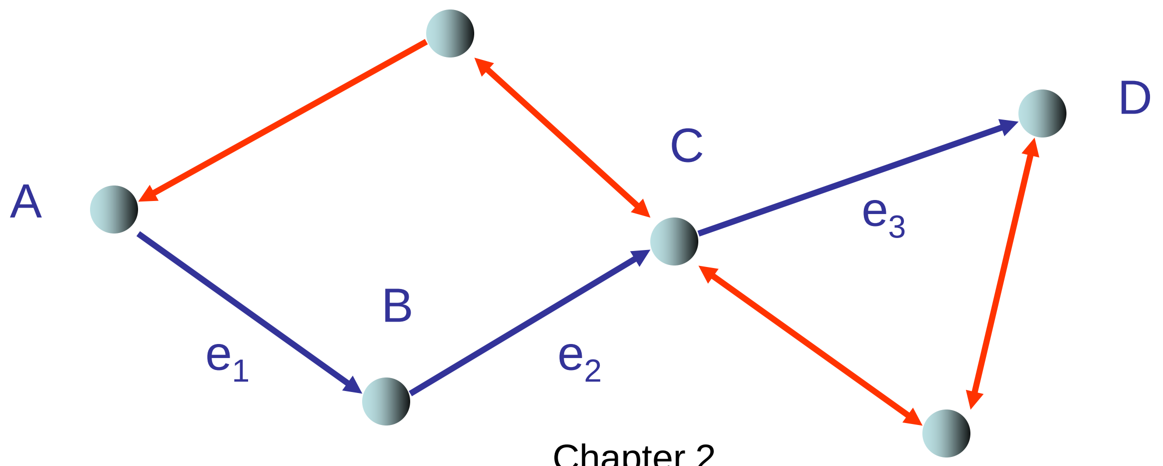
Degree of G: $\Delta = \max_{v \in V} \delta(v)$



Graph Theory

Definition 2.3: Let $G=(V,E)$ be a graph. An edge sequence $p=(e_1,e_2,\dots,e_k)$ in G is called a path if there is a node sequence $(v_0,\dots,v_k)$ with

- G undirected: $e_i=\{v_{i-1},v_i\}$ for all $i\in\{1,\dots,k\}$
- G directed: $e_i=(v_{i-1},v_i)$ for all $i\in\{1,\dots,k\}$



Graph Theory

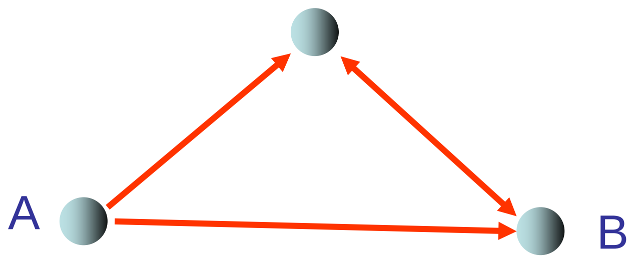
Definition 2.4: A graph $G=(V,E)$ is called

- connected if G is undirected and for all node pairs $v,w\in V$ there is a path from v to w in G.
- weakly connected if G is directed and for all node pairs $v,w\in V$ there is a path from v to w in the undirected version of G.
- strongly connected if G is directed and for all node pairs $v,w\in V$ there is a (directed) path from v to w in G.

Graph Theory

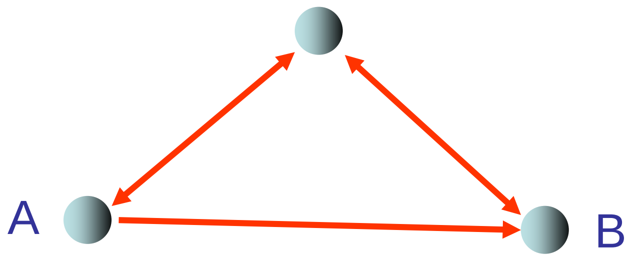
Examples:

(1) Graph is only weakly connected



no directed path
from B to A

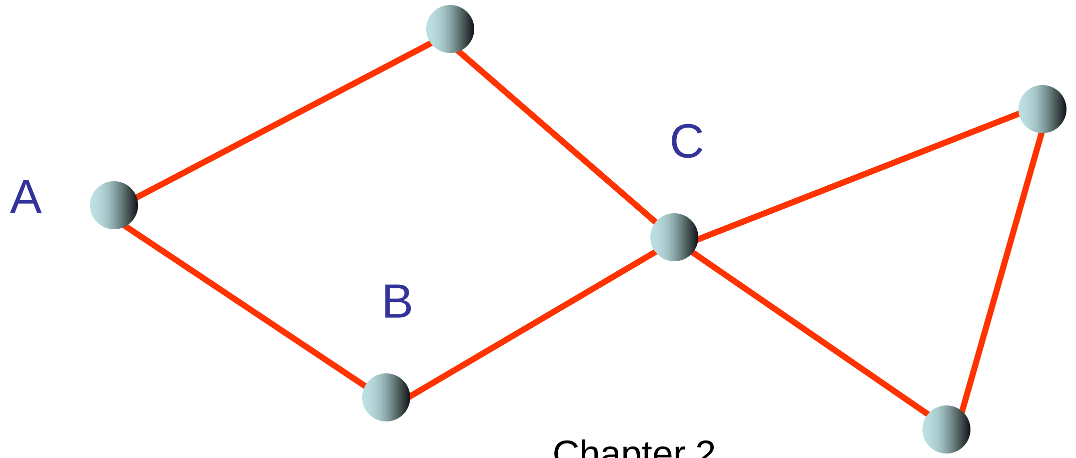
(2) Graph is strongly connected



Graph Theory

Definition 2.5: Let $G=(V,E)$ be a graph and $p=(e_1,e_2,\ldots,e_k)$ be a path from v to w in G.

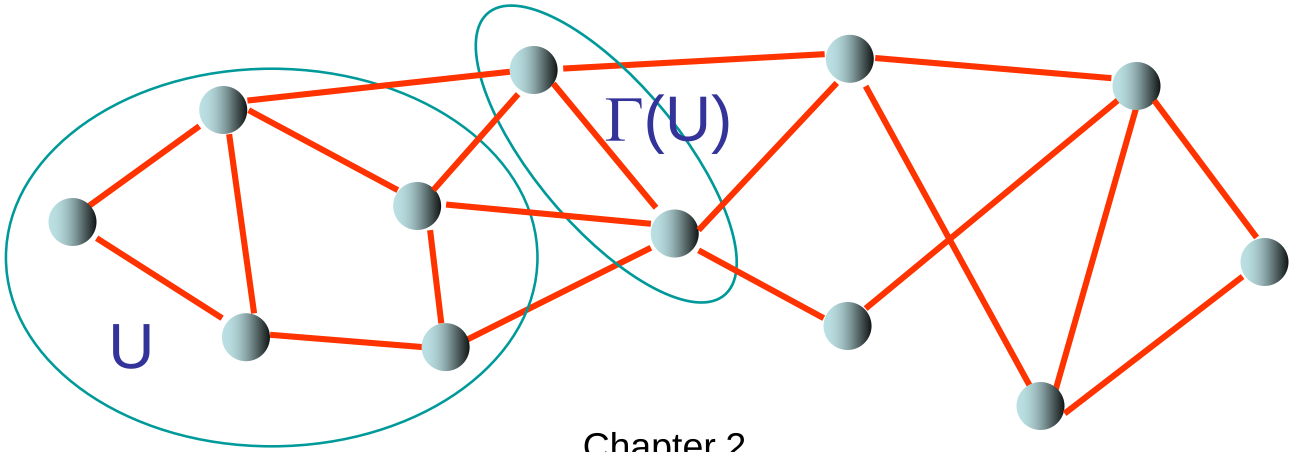
- Length of p: $|p|=k$
- Distance of w from v: $d(v,w)$ = min. path length from v to w ($d(v,w) = \infty$ if there is no path from v to w)
- Diameter of G: $D(G)=\max_{v,w\in V} d(v,w)$



Graph Theory

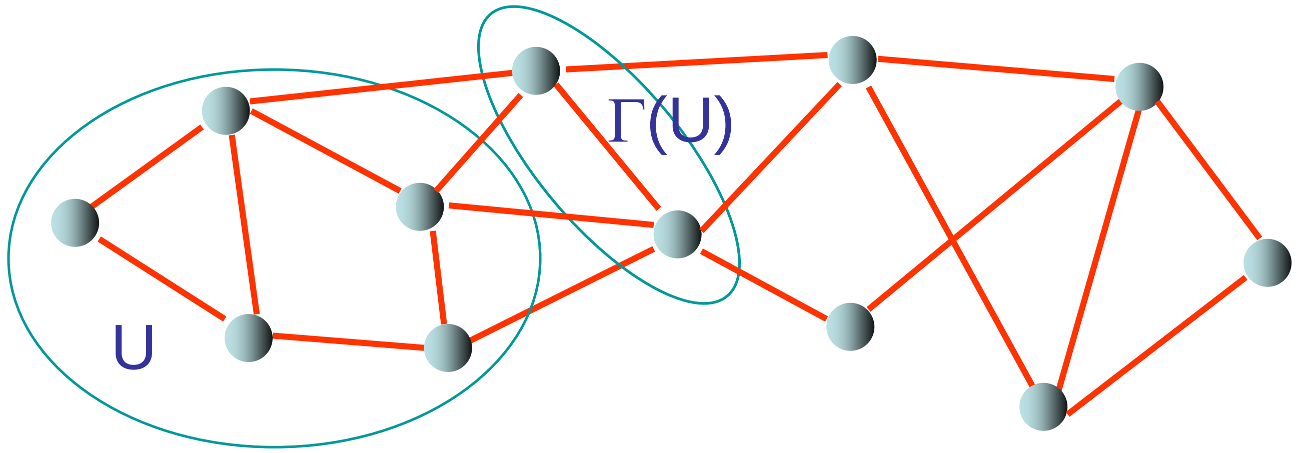
Definition 2.6: Let $G=(V,E)$ be a graph.

- $\Gamma(U)$: neighbor set of a node set $U\subseteq V$, i.e.,
  $\Gamma(U)=\{ w\in V\setminus U \mid$ there is a $v\in U$ with $\{v,w\}\in E$ (resp. $(v,w)\in E$ in the directed case) $\}$
- $\alpha(U) = |\Gamma(U)| / |U|$: expansion of U
- $\alpha(G) = \min_{U\subseteq V, 1\le|U|\le\lceil |V|/2\rceil} \alpha(U)$: expansion of G



Graph Theory

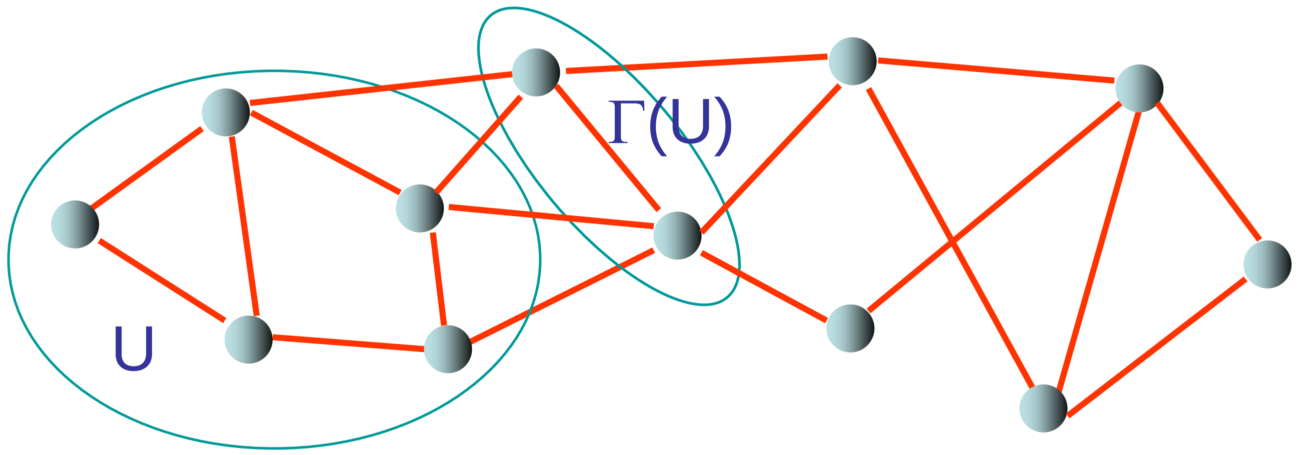
Expansion: k failures $\Rightarrow$ at most $k/\alpha(G)$ nodes get disconnected from rest of the graph



Proof: Let U be set of all non-failing nodes that get disconnected due to failed nodes. Then all nodes in $\Gamma(U)$ failed, i.e., $|\Gamma(U)| \leq k$. Moreover, $\alpha(U) \geq \alpha(G)$ and $\alpha(U) = |\Gamma(U)|/|U|$, so $|U| \leq k/\alpha(G)$.

Graph Theory

Expansion: k failures $\Rightarrow$ at most $k/\alpha(G)$ nodes get disconnected from rest of the graph

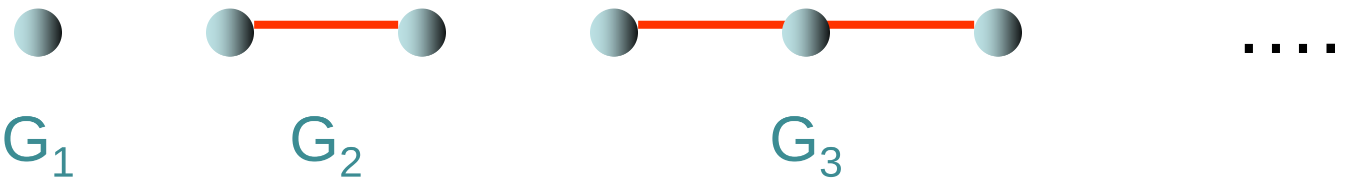


Expansion should be as high as possible

Graph Theory

In the following we consider classical families of graphs $\mathcal{G}=\{G_1, G_2, \ldots\}$.

Example: Family of linear lists

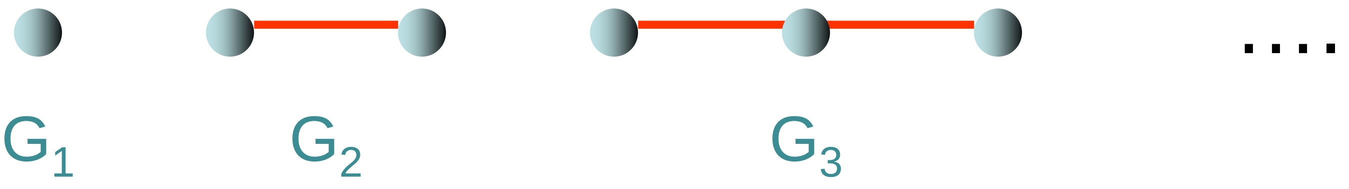


We say: graph G from a family $\mathcal{G}$ has constant degree if the degree of all graphs in $\mathcal{G}$ is upper bounded by a constant.

Graph Theory

In the following we consider classical families of graphs $\mathcal{G}=\{G_1, G_2, \ldots\}$.

Example: Family of linear lists

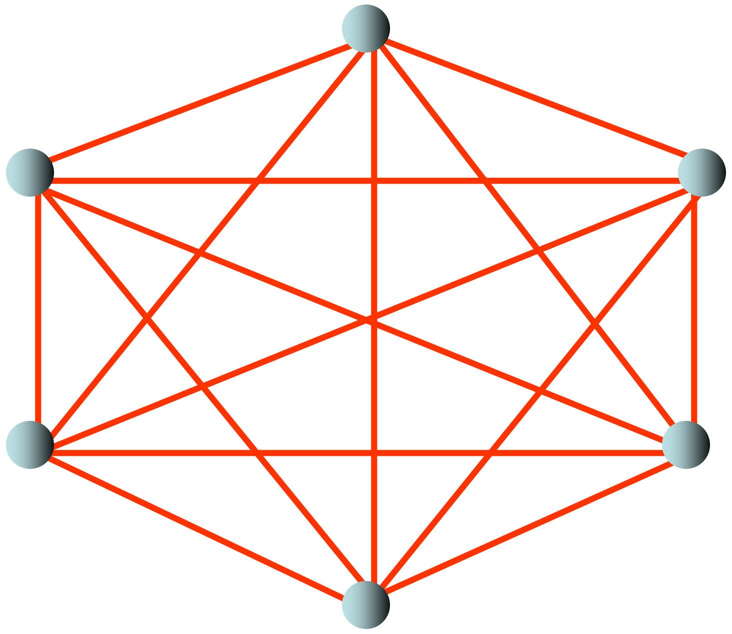


For a graph G from $\mathcal{G}$ we use

- n: number of nodes (resp. size) of G
- m: number of edges of G

Classical Graph Families

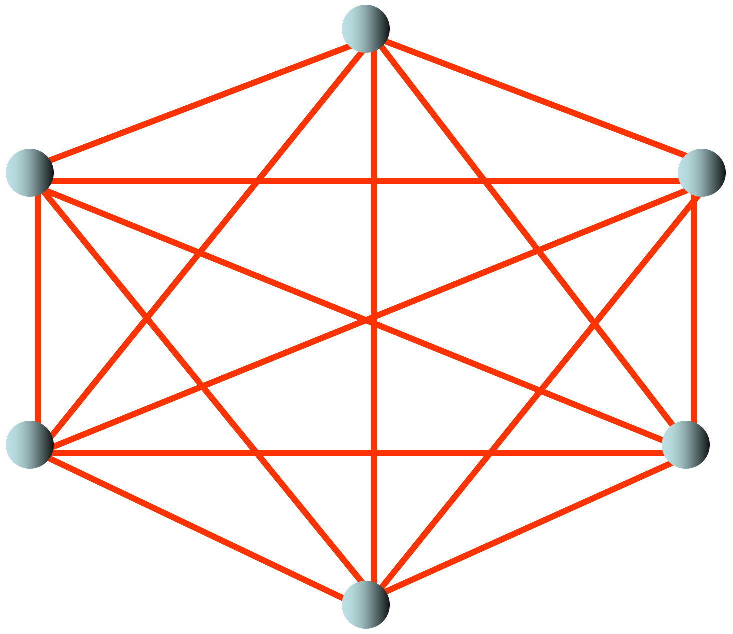
Complete graph / clique: every node is connected to every other node



Advantage: low diameter, high expansion

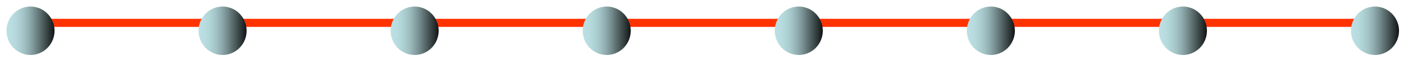
Classical Graph Families

Complete graph / clique: every node is connected to every other node



Problem: high degree! ($\delta(v)=n-1$ for all v)

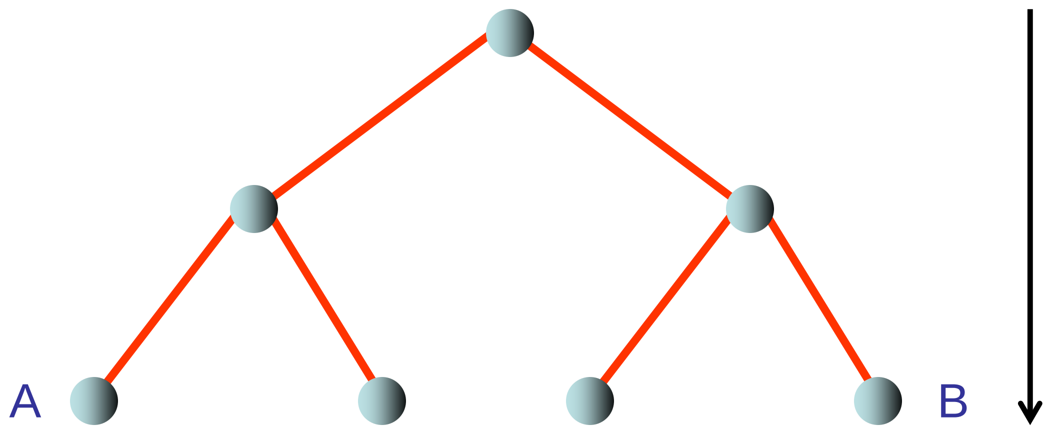
Linear List



- Degree 2 (minimal for connectivity), BUT
- Diameter is bad (D(List)=n-1)
- Expansion is bad (α(List)$\approx$2/n)

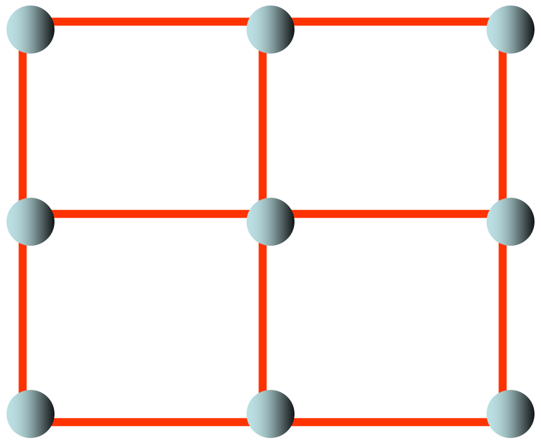
How to obtain a small degree and diameter?

Complete Binary Tree



- $n=2^{k+1}-1$ nodes when depth is $k \in \mathbb{N}_0$
- degree is 3
- Diameter is $2k \approx 2 \log_2 n$, BUT
- Expansion is bad ($\alpha(\text{tree}) \approx 2/n$)

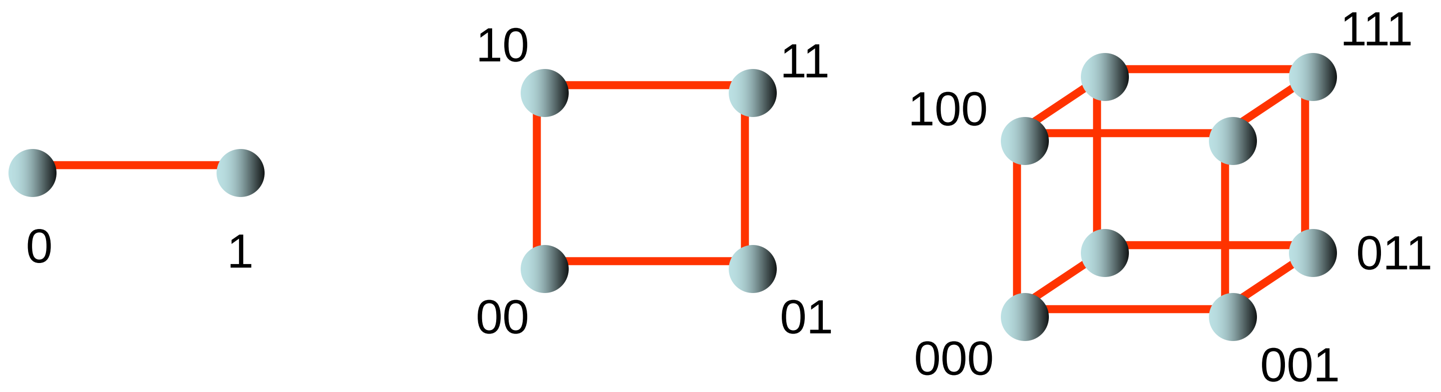
2-dimensional Grid



- $n = k^2$ nodes when there are k nodes along each side, maximal degree 4
- Diameter is $2(k-1) \approx 2\sqrt{n}$
- Expansion is $\approx 2/\sqrt{n}$
- Not bad, but can we do better?

d-dimensional Hypercube

- Nodes: $(x_1,\ldots,x_d) \in \{0,1\}^d$
- Edges: $\forall i: (x_1,\ldots,x_d) \to (x_1,..,\underset{\nwarrow \text{Bit i flipped}}{1-x_i},..,x_d)$

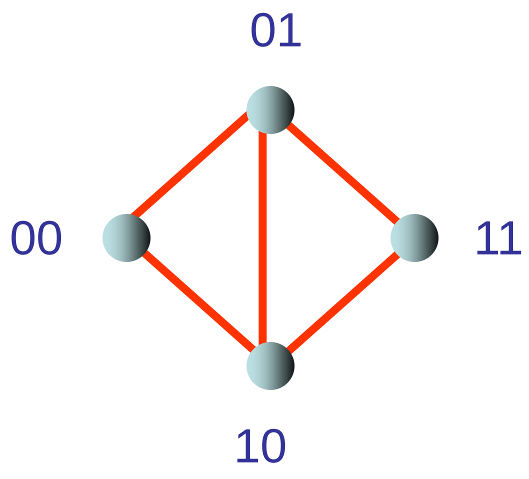


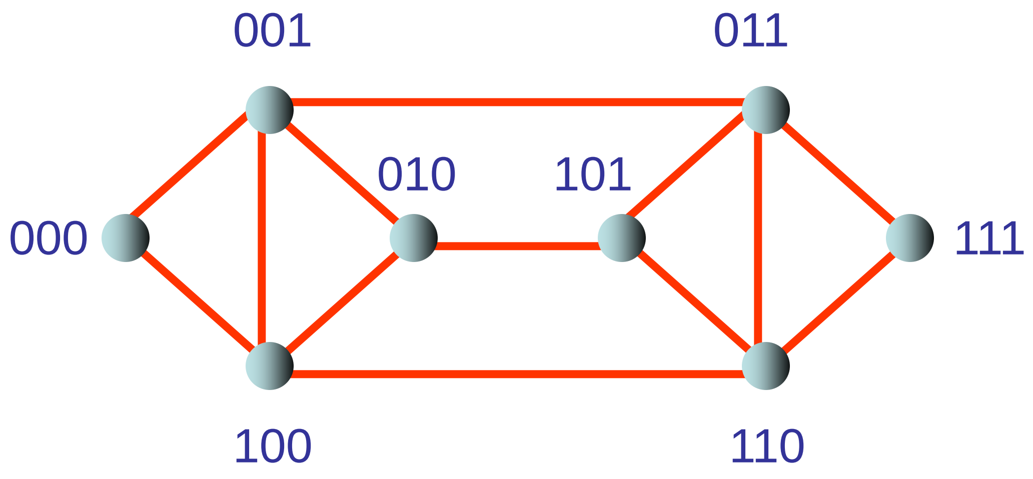
d=1 d=2 d=3

Degree d, diameter d, expansion $\approx 1/\sqrt{d}$

d-dimensional de Bruijn Graph

- Nodes: $(x_1,\dots,x_d) \in \{0,1\}^d$
- Edges: $(x_1,\dots,x_d) \rightarrow (0,x_1,x_2,\dots,x_{d-1})$
  $(1,x_1,x_2,\dots,x_{d-1})$





Diameter

Theorem 2.7: Every graph of maximal degree $\delta>2$ and size n has a diameter of at least $(\log n)/(\log(\delta-1))-1$.

Proof: exercise

Theorem 2.8: For all even $\delta>2$ there is a family of graphs of maximal degree δ and size n with diameter at most $(\log n) / (\log \delta -1)$.

Proof: exercise

Expansion

Theorem 2.9: For every graph G it holds that $\alpha(G)\in[0,1]$.

Proof: see the definition of the expansion $\alpha(G)$.

Theorem 2.10: There is a family of graphs with constant degree and constant expansion.

Example: Gabber-Galil Graph

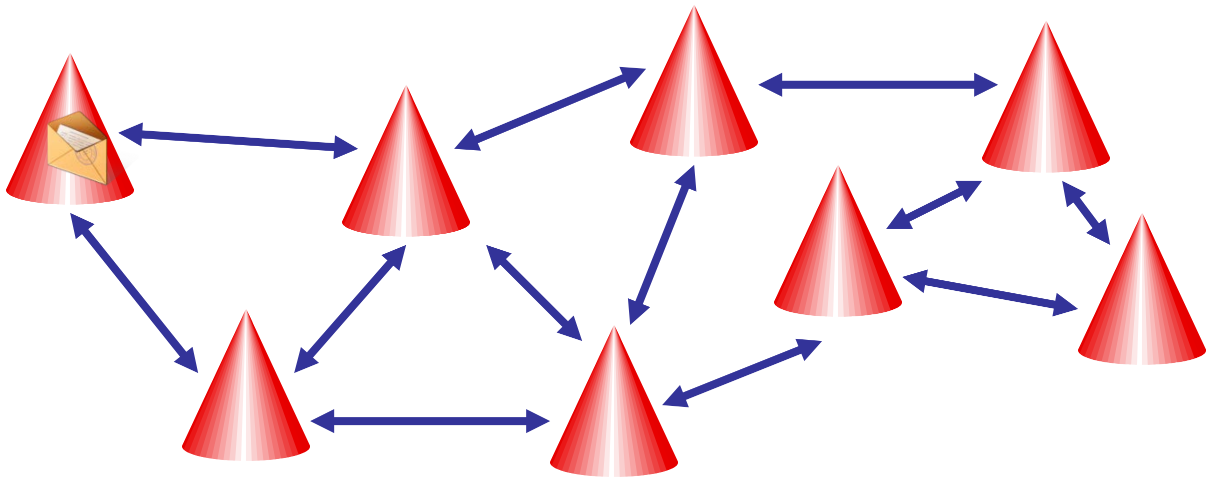
- Node set: $(x,y) \in \{0,\dots,k-1\}^2$
- $(x,y) \to (x,x+y),(x,x+y+1), (x+y,y), (x+y+1,y) \pmod k$

Overview

- Graph theory
- Classical graph families
- Fundamental graph parameters
- Process Model
- Pseudo-Code
- Synchronization

Process Model

- Processes can connect to each other



- Processes communicate via message passing

Process Model

Types of actions:

- Triggered by a local/remote call:
  $\langle$name$\rangle$($\langle$parameters$\rangle$) $\rightarrow$ $\langle$commands$\rangle$
- Triggered by a local state:
  $\langle$name$\rangle$: $\langle$predicate$\rangle$ $\rightarrow$ $\langle$commands$\rangle$

All messages are remote action calls.

Example:

```
minimum(x,y) →
        if x<y then m:=x else m:=y
        print(m)
```

Action „minimum“ is executed upon receipt of a request to call minimum(x,y). No return of values possible when called remotely to make sure processes don‘t starve while waiting for return value of another process!

Process Model

Types of actions:

- Triggered by a local/remote call:
  $\langle$name$\rangle$($\langle$parameters$\rangle$) $\rightarrow$ $\langle$commands$\rangle$
- Triggered by a local state:
  $\langle$name$\rangle$: $\langle$predicate$\rangle$ $\rightarrow$ $\langle$commands$\rangle$

All messages are remote action calls.

Example:

```
timeout: true →
        print(„I am still alive!“)
```

„true“ ensures that the action is periodically executed by the given process.

Process Model

Execution of actions: Processes can act concurrently but within a process the actions must be executed in a strictly sequential, and therefore atomic way.

$\rightarrow$ every action must eventually terminate!

This simplifies correctness proofs.

Pseudo-Code

Pseudo-code like in object-oriented programming:

```
Subject ⟨Name⟩:     // declares process type
    local variables
    actions
```

Types of actions:

```
 ⟨ActionName⟩(⟨Parameters⟩) →
      commands in pseudo-code

 ⟨ActionName⟩: ⟨Predicate⟩ →
      commands in pseudo-code
```

Special action:

```
 entry(⟨Parameters⟩) → // constructor
      commands in pseudo-code
```

Pseudocode

- Assignment via :=
- Loops (for, while, repeat)
- Conditional branching (if – then – else)
- Comment via { }
- Block structure via indentation
- Call of action via process reference v: v←act(…)
- Subject: stores reference to a process (empty reference: ⊥)
- Creation of new processes: new
  (new calls entry in process)

Programming Environment

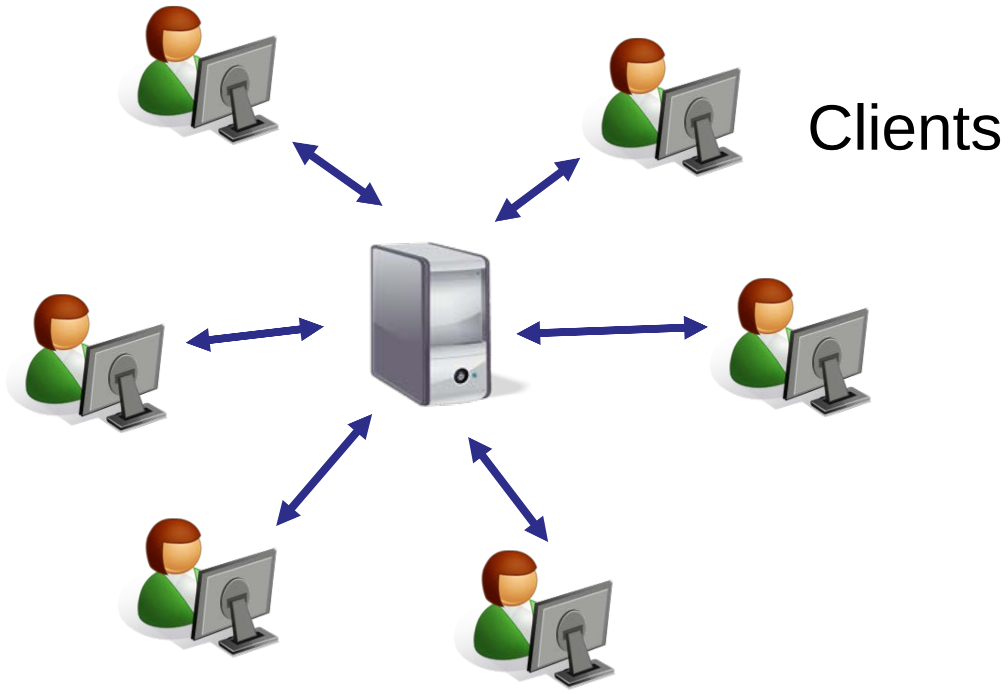
We will use a simulation environment called NetSimLan (see also the course webpage).

- Webpage: https://netsimlan.org
- Download: https://netsimlan.org/download

You can also find examples there. As you can see there, the code is very close to our pseudo-code.

Broadcasting

Simple broadcast service via server



Broadcasting

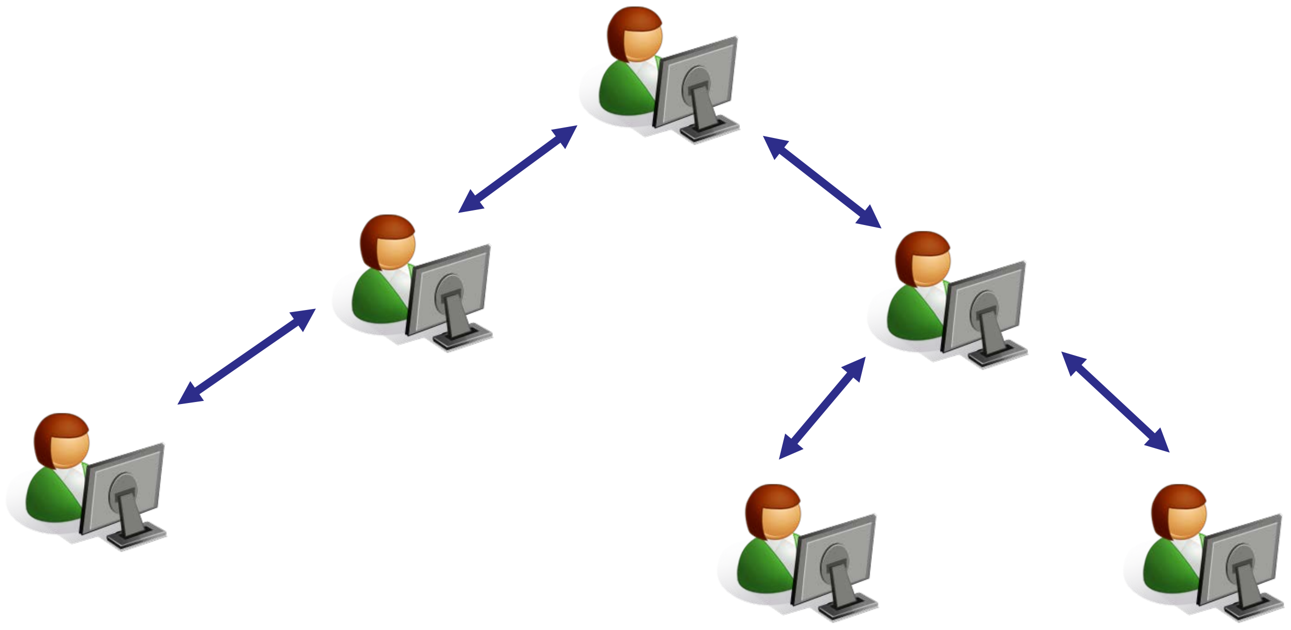
```
Subject Server:
   n: Integer                  { stores number of clients }
   Client: Array[1..MAX] of Subject   { stores refs to clients }

   entry() →               { constructor }
     n:=0

   register(C) →           { register new client with reference C }
     n:=n+1
     Client[n]:=C

   broadcast(M) → { send M to all clients }
     for i:=1 to n do
       M´:=new Object(M)     { new object containing M }
       Client[i]←output(M´)   { calls output in Client[i] }
```

Broadcasting

```
Subject Client:
  Server: Subject

  entry(S) →          { constructor }
     Server:=S        { S: reference of server }
     Server←register(this)
                           { send client ref. to server }

  broadcast(M) →           { broadcast M via server }
     Server←broadcast(M)

  output(M) →              { output M }
     print M
```

Broadcasting

Broadcast via tree



Broadcasting

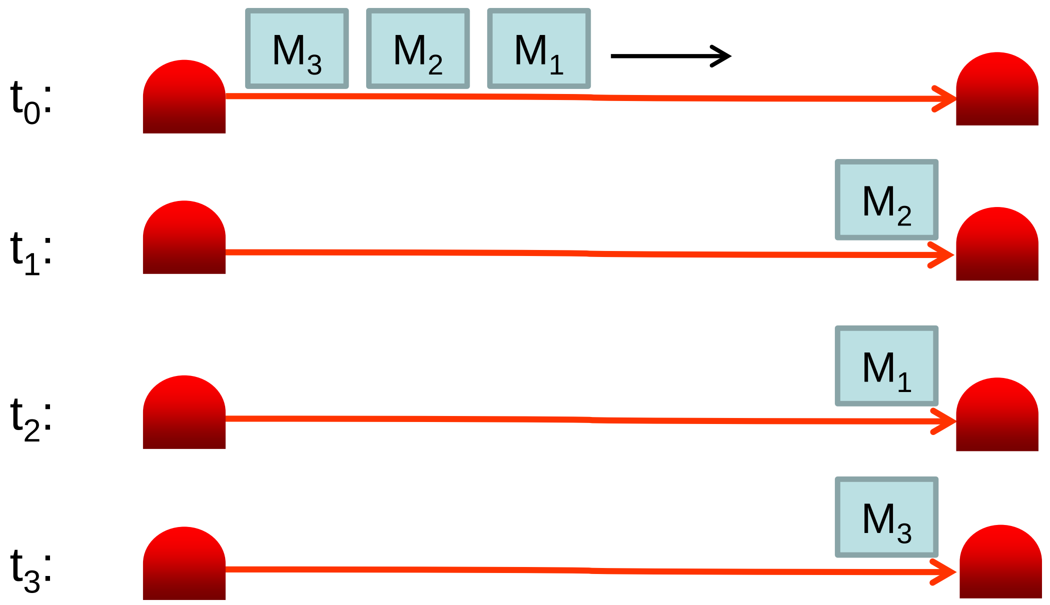
```
Subject TreeNode:
    Parent: Subject
    Child: Array[1..c] of Subject   { refs to children }

sendup(M) → { send M to root to start broadcast }
    if Parent≠⊥ then Parent←sendup(M)
                else broadcast(M)

broadcast(M) → { send M to all children }
     print M
     for i:=1 to c do
         if Child[i]≠⊥ then
             M´:=new Object(M)    { new object with M }
             Child[i]←broadcast(M´)
```

Overview

- Graph theory
- Classical graph families
- Fundamental graph parameters
- Process Model
- Pseudo-Code
- Synchronization

Synchronization

The process model assumes asynchronous message passing:



- all messages are eventually delivered
- but no time bounds and FIFO delivery guaranteed

Synchronization

Synchronous message passing model:

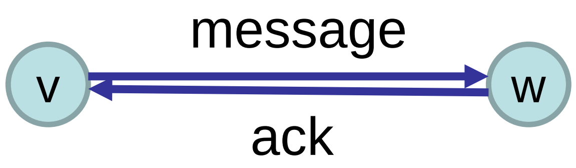
- Time proceeds in synchronous rounds (a round takes one time unit and all nodes start a round at the same time).
- Every round works as follows:
  - First, all messages sent to some node v in the previous round are received by v, and
  - then v does some local computations based on the received messages and may send out messages to some of its neighbors, which are received in the next round.

For simplicity, many of our algorithms are presented for the synchronous message passing model.

In order to emulate such algorithms in an asynchronous message passing model, we need a synchronization mechanism.

Synchronization

- A node is called safe with respect to a certain round if each message of the synchronous algorithm sent by that node at that round has already arrived at its destination.
- Safety can be checked with acknowledgements: whenever a message is received by a node, an acknowledgement is sent back to the sending node.



- A node detects that it is safe whenever all of its messages have been acknowledged.

Synchronization

Synchronizer α:

- Using the acknowledgement mechanism, each node eventually detects that it is safe and reports this to all of its neighbors.
- Whenever a node learns that all of its neighbors are safe and it is safe as well, it starts a new round.

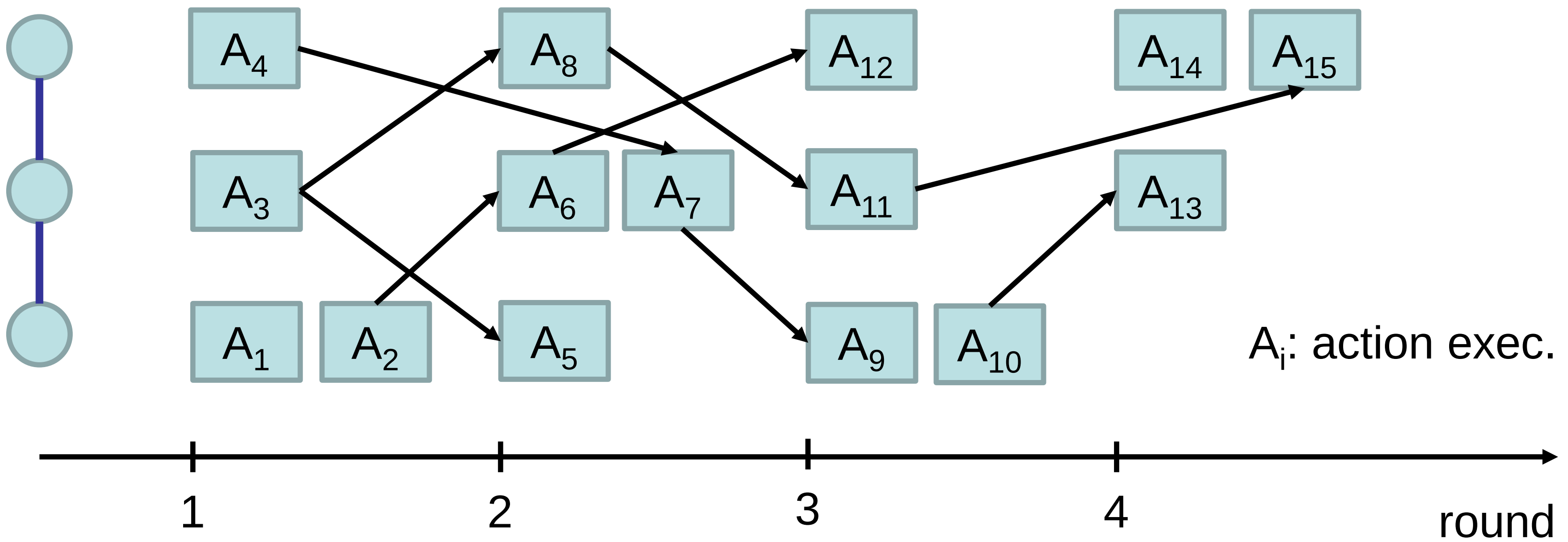
To check:

- Safety: The synchronizer correctly emulates a synchronous message passing system.
- Liveness: A node never has to wait indefinitely to start a new round.
- Time: Number of time steps for sync. of a round.
- Work: Number of messages needed for sync. of a round.

Synchronization

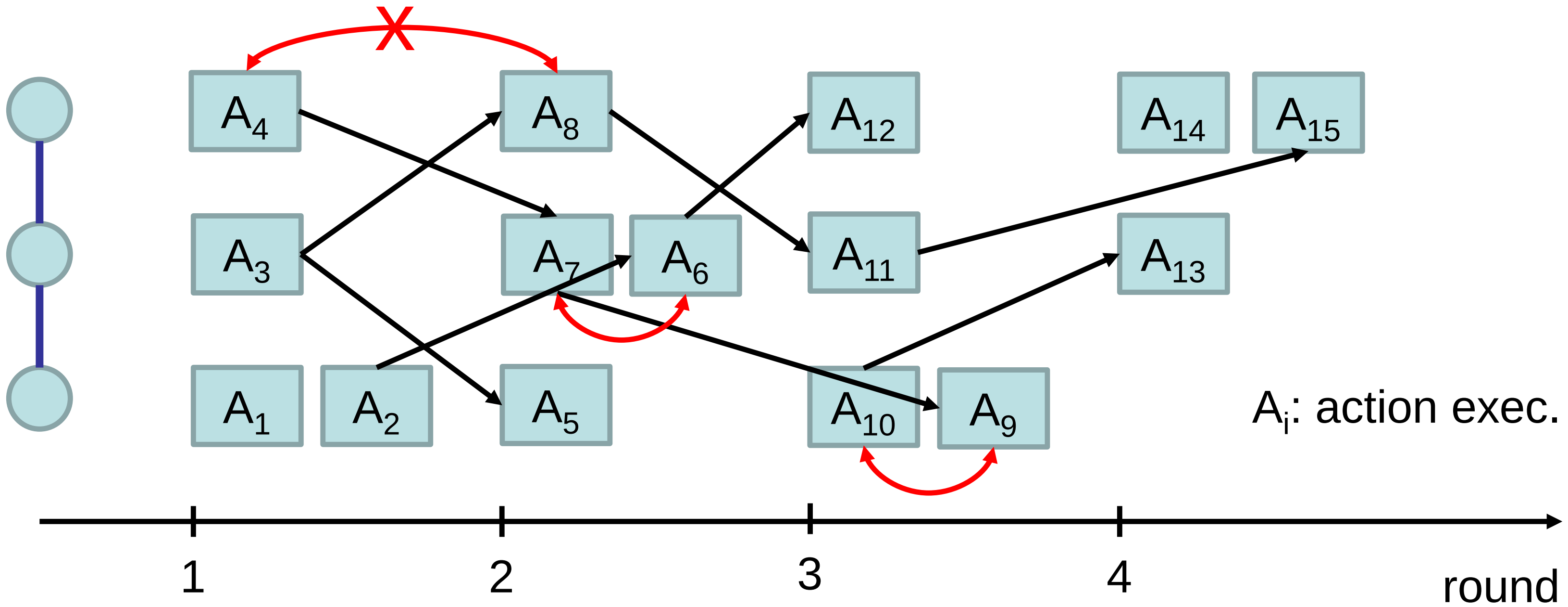
What does it mean that a synchronizer correctly emulates a synchronous message passing system?

Consider the following example ($\longrightarrow$: triggered by msg)



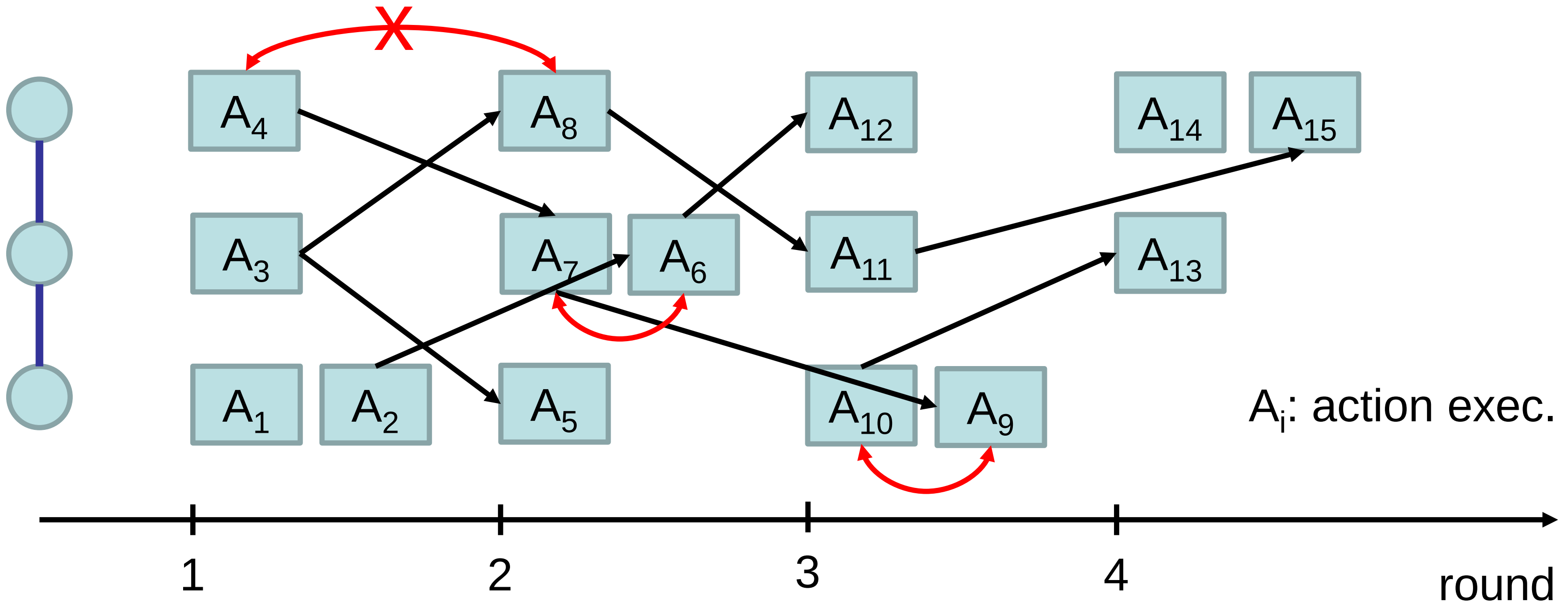
Synchronization

In synchronous message passing, we are usually allowed to reorder execution of actions in the same round, but not from different rounds.



Synchronization

Condition for any synchronizer: For every two action executions A_i, A_j in some process v, where A_i is processed before A_j, $r(A_i) \le r(A_j)$, where $r(A)$ is the (sync.) round of A.



Synchronization

Synchronizer α:

- Using the acknowledgement mechanism, each node eventually detects that it is safe and reports this to all of its neighbors.
- Whenever a node learns that all of its neighbors are safe and it is safe as well, it starts a new round.

Formal safety condition:

- $r(v)$: current round of node v (initially, $r(v)=1$ for every v)

Theorem 2.11: For every two action executions A_i, A_j in some process v, where A_i is processed before A_j, $r(A_i) \leq r(A_j)$, where $r(A)$ is the (sync.) round of A.

Proof:

We first present a pseudo-code for the synchronizer α and then provide a proof sketch for Theorem 2.11.

Synchronization

```
Subject Node:
    round: Integer                        { current round, initially set to 1 }
    Neighbor: Array[] of Subject          { array of references to neighbors }
    safe_current: Integer                 { # safe msgs from current round, initially set to 0 }
    safe_next: Integer                    { # safe msgs from next round, initially set to 0 }
    safe: Boolean                         { safety status for current round, initially set to false }
    missing_acks: Integer                 { # missing acks of current round }
    Q_current: Array[] of Message         { queue of messages that need to be processed in next round, initially ∅ }
    Q_next: Array[] of Message            { queue of messages that need to be processed in round after next round, initially ∅ }

entry() → { executes actions for round 1 and updates missing_acks}

timeout: true → { checks if there are missing acks or if all neighbors and node itself are safe }
    if missing_acks=0 and not safe then                { all requests of current round have been acknowledged: }
        safe:=true                                     { process is now safe }
        for each v∈Neighbor do                         { send out safety announcements to all neighbors }
            v←neighbor_safe(round)
    if safe and safe_current=length(Neighbor) then     { all neighbors and node itself are safe for given round: }
        round:=round+1; safe:=false                    { switch to next round }
        safe_current:=safe_next                        { move #safe-msgs from next to current round }
        safe_next:=0                                   { safe-msgs for new next round cannot have arrived at v yet }
        for each M∈Q_current do
            { process M as given by synchronous protocol and update missing_acks }
        { execute any action that is supposed to be executed in each round, and update missing_acks }
        Q_current:=Q_next; Q_next:=∅                   { new Q_next cannot have received any messages yet }

neighbor_safe(r) → { notification from neighbor that it is safe for round r, where r∈{round,round+1} }
    if r=round then safe_current:=safe_current+1
               else safe_next:=safe_next+1

process_msg(v,r,M) → { M, generated by v at round r∈{round,round+1}, needs to be processed at round r+1 }
    v←ack()                               { acknowledge receipt of message }
    if r=round then add M to Q_current    { process M in next round }
               else add M to Q_next       { process M in round after next round }

ack() → { acknowledgement received for a message sent in current round }
    misssing_acks:=missing_acks-1
```

Synchronization

Proof sketch of Theorem 2.11

From the definition of the synchronizer α it follows:

a. For every $\{v,w\}\in E$, $|r(v)-r(w)|\leq 1$ at any time.
(Proof: by induction on the receipts of safe-messages over the time.)

b. Also, a node v at round r(v) has already received safe-messages from all of its neighbors for round r(v)-1, which together with (a) implies that v can only receive safe-messages for round r(v) and r(v)+1. Thus, it is sufficient to remember the number of received safe-messages in two counters: safe_current and safe_next.

Furthermore, by induction on the number of rounds it holds:

1. Whenever v sends a safe-msg to w for round r, then all messages that v is supposed to send out in round r have already been acknowledged, which means that v will not send out any further messages to w in round r.
2. Due to (a), (b) and (1): Whenever v receives a message in round r, it must be from round r or r+1, which means that it needs to be processed (according to the synchronous message passing model) in round r+1 (which is collected in Q_current) or r+2 (which is collected in Q_next).
3. Due to (1) and (2): For every node v that enters round r+1, Q_current contains all messages that v is supposed to process at round r+1.
4. Due to (3): Whenever missing_acks=0, all of v's messages that are supposed to be received in round r(v) have been received, so that v can declare itself to be safe.

These properties imply that the actions are executed in the right order for every node v, i.e., Theorem 2.11 is correct.

Synchronization

Synchronizer α:

- Using the acknowledgement mechanism, each node eventually detects that it is safe and reports this to all of its neighbors.
- Whenever a node learns that all of its neighbors are safe and it is safe as well, it starts a new round.

- Safety: see Theorem 2.11.
- Liveness: Is guanteed since we assume that every message is eventually delivered (and thereby eventually processed).
- Time: O(max. message delay)
- Work: $O(|E|)$

Problem: work can be quite high if $|E|$ is large!

Synchronization

Synchronizer β:

- Preprocessing: nodes build up spanning tree rooted at some node r.
- Nodes report to r via a convergecast in the tree that they are safe. Convergecast: Starts at the leaves and ends at the root. Whenever a node learns that it is safe and all of its descendants in the tree are safe (which it learns from its children), it reports this to its parent.
- Once r learns that all nodes are safe, it broadcasts this to all nodes via the spanning tree.
- Once a node learns that all nodes are safe, it starts the next round.

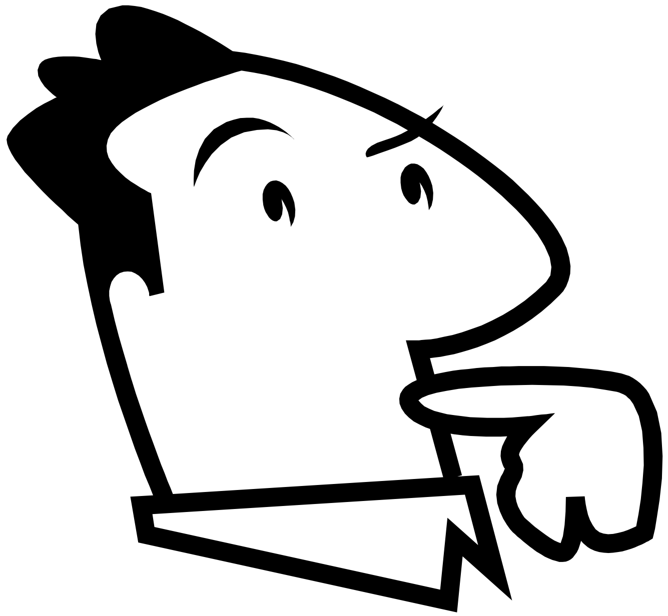
- Safety: The synchronizer β correctly emulates a synchronous message passing system. (Not shown here.)
- Liveness: A node never has to wait indefinitely to start a new round. (Guaranteed if every message is eventually delivered and therefore processed.)
- Time: $O(D \cdot (\text{max. message delay}))$ (D: depth of spanning tree)
- Work: $O(|V|)$

Synchronization

```
Subject TreeNode:
    Parent: Subject
    Child: Array[1..c] of Subject          { refs to children }
    children: Integer                      { current number of children }
    safechildren: Integer                  { initially, safechildren=0 }
    safe: Boolean                          { initially, safe=false, safe set to true externally }

timeout: true → { checks if all children and node itself are safe }
    if safechildren=children and safe then
        safechildren:=-1                   { only one converge() call made }
        if Parent=⊥ then broadcast()       { root: inform nodes that all are safe }
        else Parent←converge()             { not root: inform Parent that all below are safe }

converge() → { notification from a child that it is safe }
    safechildren:=safechildren+1

broadcast() → { informs all children that all nodes are safe }
    safe:=false                      { start new round }
    safechildren:=0
    for i:=1 to c do
        if Child[i]≠⊥ then Child[i]←broadcast()
```

References

- Baruch Awerbuch. Complexity of Network Synchronization. *Journal of the ACM* 32(4): 804-823, 1985.
- A. Bagchi, A. Bhargava, A. Chaudhary, D. Eppstein, and C. Scheideler. The effect of faults on network expansion. In *Proc. of the 16th ACM Symp. on Parallel Algorithms and Architectures (SPAA), pages 286*–293, 2004.
- O. Gabber and Z. Galil. Explicit constructions of linear-sized superconcentrators. *Journal of Computer and System Sciences, 22:407–420, 1981.*
- F. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays · Trees · Hypercubes. Morgan* Kaufmann Publishers, 1992.
- A. Lubotzky, R. Phillips, and R. Sarnak. Ramanujan graphs. *Combinatorica, 8(3):261–277, 1988.*
- C. Scheideler. *Universal Routing Strategies for Interconnection Networks.* Lecture Notes in Computer Science 1390. Springer, 1998.



Questions?