Lecture 2

Synchronizers

In the previous lecture, we have seen that results may vary quite a bit depending on the model. Today, we study one very important distributed model in more detail. We will see that it can be a good approach to first figure out how to *simulate* a more powerful model before attempting to solve a difficult problem.

2.1 Synchronous message passing

As promised in the previous lecture, we'll be more precise about the considered model(s) today. If you expect that this means tedious definitions, that's not actually true. Many useful models are useful *because* they are simple, implying that understanding them means to learn something about a wide variety of systems. This is also the case for the following well-studied model of distributed computing.

Definition 2.1 (The LOCAL model)**.** *The network is modeled as an undirected, simple graph $G = (V, E)$ of n nodes. Each node has a unique identifier of $\mathcal{O}(\log n)$ bits. An algorithm is executed in* synchronous rounds*, where in each round, each node (a.k.a. processor) takes the following steps:*

1. *Do some local computations.*
2. *Send messages to its neighbors in the graph G.*
3. *Receive messages (that were sent by neighbors in step 2 of the same round).*

In addition, nodes may determine a (local) output and terminate at the end of a round. The time complexity *of a synchronous message passing algorithm is the time until all nodes have terminated. Finally, some problems will also have some additional input information, e.g., edge weights. Nodes will then initially know the local part of the input, e.g., the weight of incident edges.*

Clearly, this model is questionable for many reasons:

- We do not put restrictions on local computations or memory, which means that nodes could solve NP-hard problems locally! Some authors consider this cheating and require computations polynomial in n.

- Nodes can send messages of arbitrary size. In particular, they simply can send everything the know to all neighbors in each round. This can easily result in unrealistically large messages of size $\Omega(n^2)$, e.g., in a complete graph.
- We assume perfectly synchronous execution, but many practical systems simply cannot operate this way or, if forced to do so, would have extremely long rounds (because one needs to wait for the slowest computations finish and all messages to arrive before moving on to the next round).

So, why is the LOCAL model useful at all? First of all, it was originally designed for proving *lower bounds*, just like the one given in Theorem 1.7. This makes the statement only more powerful: even if all the above crazy things were possible, it would *still* take at least $1/2 \cdot \log^* n - \mathcal{O}(1)$ rounds to color the list with few colors. Such results are surprisingly useful, simply because the tell us what we *shouldn't* spend our time on trying.

Second, algorithms designed in the LOCAL model frequently turn out to use very little computations and memory, as well as small messages. The Cole-Vishkin coloring algorithm is a prime example for this. After all, only so much can be done with little information! Hence, it's not a bad approach to design a (fast) algorithm in this model and worry about things like message size later.

Finally, even if the designed algorithms are fast (in terms of the number of rounds), but use large messages, we can either try to get rid of the "cumbersome" aspects of the algorithm or know that an algorithm being slow must be caused by a limitation in bandwith or computation.

2.2 Asynchronous message passing

Still, all that doesn't address the issue of synchrony, which, to put it mildly, turns out to be more of a problem. Consequently, there is a "sister" to the LOCAL model, the asynchronous message passing model.[1]

Definition 2.2 (The asynchronous message passing model). *The network is modeled as an undirected, simple graph $G = (V, E)$ of n nodes. Each node has a unique identifier of $\mathcal{O}(\log n)$ bits. An algorithm is executed based on* events. *An* event at node *$v \in V$ is either the node starting to execute the algorithm or receiving a message from a neighbor (nodes start the algorithm at the latest on reception of the first message). Upon an event at node v, v does the following:*

1. *It does some local computations.*
2. *It may send messages to its neighbors in the graph G.*

Each sent message will eventually be received, but it is completely arbitrary which of the messages in transit will arive next. Just like before, a node may also determine a (local) output and terminate upon an event.

Observe that an asynchronous algorithm can also operate in the synchronous model: If all nodes start the execution at time 0 and each message is under way

[1]The LOCAL model is also called *sychronous message passing* model, but that's a mouthful and I'm lazy.

for exactly 1 time unit, this is exactly the same as executing the algorithm in a synchronous system.

It is a crucial aspect of the asynchronous model that the time for a message to reach its destination is unbounded. However, we still would like to figure out what algorithms are "fast" in this model. We do this by giving the algorithm more slack.

Definition 2.3 (Asynchronous time complexity). *An algorithm in the asynchronous message passing model has time complexity T, if in all executions in which all nodes start the algorithm at time* 0 *and each message is received at most one time unit after it was sent, all nodes terminate by time T.*

Remarks:

- An asynchronous algorithm of time complexity T has also synchronous time complexity T.
- One can extend this definition to allow for not all nodes "waking up" at time 0.
- Assuming asynchrony is typically unrealistic, too. However, this time we're overly pessimistic, which means that algorithms can deal with "more synchronous" models, while lower bounds cannot.
- A lower bound or impossibility result based on asynchrony thus means that one can/should look for adding constraints that make the system "more synchronous", enable better algorithms, and yet are still pessimistic enough to be realistic.
- The synchronous and asynchronous models are two extremes, so understanding them also helps understanding what's in between.

2.3 Simulating synchrony

Our goal is to be able to "pretend" that the system is synchronous when coming up with algorithms. In other words, we would like to figure out a (generic) way of transforming a synchronous algorithm into an asynchronous one. The new algorithm should behave just like the old, which is captured by the following definition.

Definition 2.4 (Simulation). *Algorithm $\mathcal{A}$* simulates *algorithm $\mathcal{B}$, if, given the same inputs, both algorithms compute the same outputs.*

A *synchronizer* generates sequences of *clock pulses* at each node of the network satisfying the condition given by the following definition.

Definition 2.5 (valid clock pulse). *Assume that (upon an event), node v can "trigger clock pulse i", for $i \in \mathbb{N}$ and nodes simulate a synchronous algorithm $\mathcal{A}$. When generating clock pulse i, node v will send all messages it would send in round i according to $\mathcal{A}$; to each of these messages it will attach the round number i. Clock pulse i is* valid *if it is generated after v received all the messages of the synchronous algorithm sent to v by its neighbors in rounds $j < i$ (and has not been generated before).*

Given a mechanism that generates valid clock pulses $1, \dots, T+1$ at all nodes, a T-round synchronous algorithm can be simulated: each node can compute its output based on its local history of messages and clock pulses.

Lemma 2.6. *If all generated clock pulses are valid according to Definition 2.5 and all non-terminated nodes keep generating pulses, we can simulate the respective synchronous algorithm.*

Proof. Suppose synchronous Algorithm $\mathcal{A}$ terminates in T rounds at node v. When v generates pulse $T+1$, it has received all messages that its neighbors send during the execution of $\mathcal{A}$, neatly labeled by their round numbers. It can thus locally perform exactly the computations that $\mathcal{A}$ would, output the result, and terminate. □

Note that the messages a synchronous algorithm sends in round i are simply the "output" generated by executing the algorithm partially up to round $i-1$; if $i-1$ is 0, this just means to compute the first set of messages based on the input. In particular, it's safe to generate pulse 1 right away at each node. In other words, all we need to do is to provide a method that generates clock pulse $i+1$ at each node provided that clock pulse $i \in \mathbb{N}$ has been generated at each node and the respective messages of $\mathcal{A}$ have been sent.

The main issue with generating clock pulse $i+1$ at a node v is that (in general) v cannot know which of its neighbors sent a message in round i of $\mathcal{A}$. As messages may be in transit arbitrarily long, this means that if it produces a clock pulse without hearing from some neighbor, it risks generating an invalid pulse. On the other hand, if there is no such message, but it plays things safe, it may wait indefinitely and we have a *deadlock*.

The most simple solution to this dilemma is to make sure that there is *always* a message from *each* neighbor in *each* round. Denote by $m_{\mathcal{A}}(v, w, i)$ the message v sends to w in round i when executing $\mathcal{A}$; if $\mathcal{A}$ sends no such message or has already terminated at v, we write $m_{\mathcal{A}}(v, w, i) = \perp$. With this notation, this strategy is cast into Algorithm 5

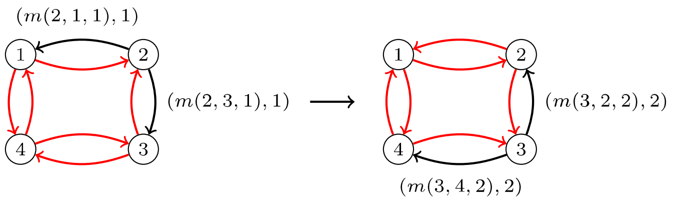

Figure 2.1: First two "rounds" of an execution of the α-synchronizer. Black arrows correspond to messages sent by the original algorithm, while red edges indicate $(\perp, i)$ messages. Actually, one can just send $\perp$ messages, where the receivers count the number of $\perp$ messages received from each neighbor; for this reason, we refrained from depicting the message contents for red arrows.

Algorithm 5 α-synchronizer simulating $\mathcal{A}$ (code for node $v \in V$)

```
if $v$ just woke up then
    $\text{pulse}_v := 1$
    for $\{v,w\} \in E$ do
        $\text{term}_v(w) := \infty$
        compute $m_{\mathcal{A}}(v,w,1)$
        send $(m_{\mathcal{A}}(v,w,1),1)$ to $w$
    end for
end if
if $v$ received $(m_{\mathcal{A}}(w,v,i),i)$ from $w$ then
    store $(m_{\mathcal{A}}(w,v,i),i)$
end if
if $v$ received $(\text{term},i)$ from $w$ then
    $\text{term}_v(w) := i$
end if
if $v$ stores $(m_{\mathcal{A}}(w,v,i),i)$ for each $\{v,w\} \in E$ with $\text{term}_v(w) < i$ and
  $\text{pulse}_v = i$ then
    $\text{pulse}_v := i+1$
    check if $\mathcal{A}$ terminates at $v$ at the end of round $i$ (from stored messages,
    ignoring $\perp$ and term)
    if $\mathcal{A}$ terminates at $v$ at the end of round $i$ then
        for $\{v,w\} \in E$ with $\text{term}_v(w) > i+1$ do
            send $(\text{term}, i+1)$ to $w$
        end for
        compute output of $\mathcal{A}$ (from stored messages) and terminate
    else
        for $\{v,w\} \in E$ with $\text{term}_v(w) > i+1$ do
            compute $m_{\mathcal{A}}(v,w,i+1)$ (from stored messages)
            send $(m_{\mathcal{A}}(v,w,i+1),i+1)$ to $w$
        end for
    end if
end if
```

Theorem 2.7 (Synchronizer α)**.** *Given a synchronous Algorithm $\mathcal{A}$ of running time T, synchronizer α simulates it in an asynchronous system with a running time of T. The number of additional messages send compared to an execution of $\mathcal{A}$ is at most* $2(T+1)|E|$.

Proof. To prove simulation, we will show that

1. All generated pulses are valid.
2. If node v terminates at the end of round T_v in $\mathcal{A}$, it generates pulses $1,\ldots,T_v+1$, terminates when generating pulse T_v+1, and outputs the correct result.
3. For each pulse $i \in \{1,\ldots,T_v\}$, v sends $(m_{\mathcal{A}}(v,w,i),i)$ to all neighbors upon generating pulse i.
4. In pulse T_v+1, v sends (term,T_v+1) to all non-terminated neighbors.

5. If, for $\{v, w\} \in E$, w terminates with pulse T_w+1, v will wait for a message from w in rounds $1, \dots, T_w + 1$.

We prove this by induction. The base case is $i = 1$. Clearly, all nodes generate valid pulse 1 and compute and transmit the messages for round 1 upon wake-up; they wait for messages from all neighbors w, since initially $\text{term}_v(w) = \infty$. Also, note that in the event that a node would terminate immediately according to $\mathcal{A}$ (i.e., "after 0 rounds"), it also does so here, computes the same output, and sends a $(\text{term}, 1)$ message to each neighbor.

Now suppose all statements are true for pulses $1, \dots, i$ of each node and consider pulse $i + 1$. As all messages for round i were computed and sent, eventually they arrive. Neighbors not sending a message have, by the induction hypothesis, terminated in an earlier round, so v set $\text{term}_v(w) = j$ for some $j < i$ and will not wait for such a message. Hence, eventually each v will generate pulse $i + 1$. As the content of all messages for pulses $1, \dots, i$ was correct, nodes correctly store them and compute and send the messages $(m_{\mathcal{A}}(v, w, i+1), i+1)$. Likewise, termination and corresponding outputs are determined correctly. This completes the induction step and thus the proof.

Concerning the time complexity, observe that all messages for pulse/round 1 are sent at time 0 and received by time 1 (assuming delays of at most 1). Hence all (non-terminated) nodes generate pulse 2 by time 1 and the corresponding messages are received by time 2, and so on. By time T, all nodes terminated or generated pulse $T + 1$, the latter also implying that they terminated because they completed T rounds of $\mathcal{A}$.

Regarding the number of sent messages, note that node v sends in pulses $i \in \{1, \dots, T_v + 1\}$ at most one message over each incident edge. Denoting by $\delta_v := |\{w \in V \mid \{v, w\} \in E\}|$ the *degree of* v, the total number of messages is at most

$$\sum_{v \in V} \sum_{i=1}^{T_v+1} \delta_v = \sum_{v \in V} (T_v + 1)\delta_v \leq (T + 1) \sum_{v \in V} \delta_v = 2(T + 1)|E|,$$

where the last equality uses that each edge has exactly two endpoints. □

Remarks:

- Despite its length, this proof is quite simple. The most difficult part is to figure out all the conditions that must be satisfied to perform the induction step and make them part of the induction hypothesis (which then makes things tedious).

- The same problem transpired when I wrote the pseudo-code for Algorithm 5. It's idea is straightfoward, but I made several mistakes. Allowing for all the cases and treating them properly can be tiresome, and makes it challenging to show correctness of more involved asynchronous algorithms.

- Fortunately, Theorem 2.7 shows that we have to do it only once! We now can devise synchronous algorithms and make them into asynchronous algorithms using the synchronizer.

- The argument extends to randomized algorithms in the following way. Interpret a randomized algorithm as a deterministic algorithm in which each node has an additional input: a (sufficiently long) string of independent, unbiased random bits. Now synchronizer α simulates a randomized synchronous algorithm in the sense that if the randomness given to each node is the same (i.e., the strings are the same), it produces the same output in the same number of "asynchronous rounds" at each node.

- Note that this can be subtle: if in a program one calls a standard function producing a random value, it may compute the "random" value by taking into account the system clock's time!
- Of course, that's not the end of the story. This equivalence between asynchronous and synchronous systems breaks down if we take into account other factors, such as the number of messages sent by an algorithm (we'll look into this now) or the possibility of failures (we'll look into this next lecture).

2.4 Synchronizing globally

Synchronizer α is "expensive" in terms of messages. This may be fine if the simulated algorithm also sends a lot of messages, implying that there's not much of a difference. Similarly, if $\mathcal{A}$ communicates only over a subset of the edges that are known in advance (e.g. a spanning tree), we can also run the synchronizer on the induced subgraph only. However, if neither is the case and we care about the number of messages, we might want to look for something else.

An obvious way of reducing the number of messages per round related to the synchronizer is to communicate control messages over fewer edges. Since we need to make sure that all (potential) neighbors are synchronized, the respective edge set must connect all nodes. A connected graph with the fewest number of edges is a tree.

Definition 2.8 (Distributed representation of a rooted tree). *A distributed* rooted tree *is defined as follows. There is a distinguished* root *node* $v_0 \in V$. *Each* $v \in V \setminus \{v_0\}$ *has a neighbor* p_v *as* parent*;* p_v *is known to* v.

Note that it's easy to root an unrooted tree (i.e., one where nodes don't know p_v) at a node v_0 in time equal to it's depth with r as root, by sending messages "down" the tree. If we also need to figure out which node becomes root, we can use, e.g., the node with largest identifier. We then start the rooting procedure at each node, including the corresponding identifier into each message, and always let the currently largest known identifier "win".

How do we use a given rooted tree for synchronization? We let the root orchestrate the execution of the algorithm. Nodes will send their messages for a given round, wait for acknowledgements from the recipients, and then consider themselves "safe" for the current round.

Definition 2.9 (Safe Node). *A node* v *is* safe *with respect to a certain clock pulse if all messages of the synchronous algorithm sent by* v *in that pulse have already arrived at their destinations.*

If all nodes are safe, we can move on to the next round. So, we let the root know when all nodes are safe and then, in turn, distribute the information that this is the case to all nodes – both via the tree. The details of synchronizer β are given in Algorithm 6.

Theorem 2.10 (β-synchronizer)**.** *Denote by d the depth of the tree employed by Algorithm 6. The algorithm simulates the synchronous T-round Algorithm $\mathcal{A}$ in $\mathcal{O}(d(T+1))$ asynchronous rounds. In total $\mathcal{O}(M+(T+1)n)$ messages are sent, where M is the number of messages sent by $\mathcal{A}$.*

Proof sketch. We give the main idea of the proof here; working out the details is similar to the proof of Theorem 2.7.

When the root issues a new pulse by sending a pulse message, it is forwarded to each node in the tree, as each node sends it to its children upon reception. Hence, if the root issues a pulse, eventually all nodes issue the pulse. Upon a pulse, nodes send the their messages for the respective round of $\mathcal{A}$, wait for the acknowledgements, and then set their safe-variable for the pulse to **true**. Note that this will eventually happen (no message is lost, acknowledgements are always sent), and it implies that the respective node is indeed safe. A safe node will send a safe message to its parent as soon as it received safe messages from all of its children. Thus, by induction on decreasing distance from the root (i.e., starting at leaves), all nodes will send safe messages to their parents; the induction also shows that this entails that the subtree rooted at the respective node consists of safe nodes only. Consequently, eventually the root will generate the next pulse, and at this point all nodes are safe.

This way the algorithm will proceed until all nodes have determined that $\mathcal{A}$ has locally terminated. This information is forwarded to the root in a similar fashion to the information that nodes are safe (using the done messages), the only difference being that not necessarily all nodes terminate in the same simulated round of $\mathcal{A}$. However, all nodes participate in the synchronizer until the root initiates the distribution of term messages, which happens when it learns that all nodes' simulation of $\mathcal{A}$ has locally terminated. We conclude that Algorithm 6 simulates $\mathcal{A}$.

Now consider the time complexity. Suppose pulse i starts at time t. By time $t+d$, all nodes have received their (pulse, i) message. Thus, by time $t+d+2$, the messages of $\mathcal{A}$ for round i have been received, acknowledged, and also these acknowledgements have arrived. Now, starting from the leaves, we see that by time $t+2d+2$, the root will have received safe messages from all its children and issue pulse $i+1$. Since $\mathcal{A}$ terminates in T rounds, all nodes will notice this when generating pulse $T+1$. Then it takes at most d time until the root learns that all nodes have completed their part of the execution of $\mathcal{A}$ and another d time to terminate, for a total of $\mathcal{O}(dT+d)=\mathcal{O}(d(T+1))$ asynchronous rounds.

Finally, let us check the message complexity. In each pulse, over each tree edge a pulse and a safe message is sent. We also have one done and one term message per tree edge. All other messages are either messages of $\mathcal{A}$ or acknowledgements for such messages. Hence, the total number of messages is

$$\begin{aligned}
&\sum_{\text{tree edges}} 2+\sum_{i=1}^{T+1}\sum_{\text{tree edges}} 2+\sum_{\text{messages of }\mathcal{A}} 2\\
=\ & 2(n-1)+2(T+1)(n-1)+2M\\
\in\ & \mathcal{O}(M+(T+1)n),
\end{aligned}$$

Algorithm 6 β-synchronizer simulating $\mathcal{A}$ (code for node $v \in V$). For simplicity, we adopt the convention that v stores all received information for later reference and performs necessary computations.

```
1: **if** $v$ just woke up **then**
2:    $\text{done}_v := \textbf{false}$
3:    **if** $v$ is root **then**
4:       send $(\text{pulse}, 1)$ to self (i.e., execute code for "received $(\text{pulse}, 1)$")
5:    **end if**
6: **end if**
7: **if** received $(\text{pulse}, i)$ **then**
8:    $\text{pulse}_v := i$
9:    $\text{safe}_v(i) := \textbf{false}$
10:   send $(\text{pulse}, i)$ to children
11:   **for** $\{v, w\} \in E$ with $m_{\mathcal{A}}(v, w, i) \neq \bot$ **do**
12:      send $(m_{\mathcal{A}}(v, w, i), i)$ to $w$
13:   **end for**
14: **end if**
15: **if** $v$ received $(m_{\mathcal{A}}(w, v, i), i)$ from $w$ **then**
16:   send $(\text{ACK}, i)$ to $w$
17: **end if**
18: **if** $v$ received $(\text{ACK}, i)$ from all $w$ with $m_{\mathcal{A}}(v, w, i) \neq \bot$ **then**
19:   $\text{safe}_v(i) := \textbf{true}$
20: **end if**
    // do the following only once for each pulse $i$
21: **if** $\text{safe}_v(i) = \textbf{true}$ and $v$ received $(\text{safe}, i)$ messages from all children **then**
22:   **if** $v$ is the root **then**
23:      send $(\text{pulse}, i+1)$ to self (i.e., execute code for "received $(\text{pulse}, i+1)$")
24:   **else**
25:      send $(\text{safe}, i)$ to parent
26:   **end if**
27:   **if** $\mathcal{A}$ terminates at $v$ at the end of round $i$ **then**
28:      $\text{done}_v := \textbf{true}$
29:   **end if**
30: **end if**
31: **if** $\text{done}_v = \textbf{true}$ and received done from all children **then**
32:   **if** $v$ is root **then**
33:      send term to all children
34:      compute output and terminate
35:   **else**
36:      send done to parent
37:   **end if**
38: **end if**
39: **if** $v$ received term **then**
40:   send term to children
41:   compute output and terminate
42: **end if**
```

provided that no node ever sends a message related to pulses larger than $T+1$.

To achieve this, we bundle up the synchronizer-related messages a node sends to its parent (i.e., safe and done) in each pulse. This means that the root cannot learn that pulse $T+1$ is complete without also noticing that $\mathcal{A}$ has terminated. The root then will just send the term message and terminate without issuing another pulse. □

Remarks:

- The last paragraph of this proof highlights again how careful one needs to be with asynchrony. If safe and done messages are handled independently, it could happen that a term message is incredibly slow and we execute a huge number of pointless pulses!

- Strictly speaking, the theorem is therefore about a slightly different version of Algorithm 6. Since this means the theorem is technically wrong, I hope that this way of presenting it is at least very didactic.

- Note that nodes cannot terminate just because they're done with simulating $\mathcal{A}$. They have to route information to and from the root.

- The β-synchronizer is an example of repeated use of the *flooding/echo* routine. The root "floods" the information to do something through the tree, and the result is then collected by the "echo" emanating from the leaves. Here, the job is to make sure that all messages of $\mathcal{A}$ have arrived, so some additional waiting can be involved.

- Flooding/echo is extremely useful when sufficient time is available (i.e., we don't mind waiting for d time). One can use the principle for detecting termination of algorithms or determining sums (including the number of nodes), averages, maxima, etc., and then making the result known to everyone.

- One can do the flooding also without having a tree at hand, instead constructing it "on the fly". In this version, the first received message determines the parent, and one speculatively sends a message to all neighbors at this point, as they might become children. This costs $|E|$ messages. If we start from a "clean slate" (no knowledge of the network topology), this number of messages is necessary to make sure that no node is missed: an unexplored edge may lead to a node with no other connection to the network.

- Surely, synchronizer β is message-optimal (up to constants)? Nope! We *could* just collect and make known the entire graph (including identifiers and inputs) to each node by collecting and distributing it using echo/flooding on the tree, using $\mathcal{O}(n)$ messages. Afterwards, each node can simulate the complete algorithm locally!

- "That's cheating!!" you might exclaim. Indeed it is, because if we could do things centrally, then we wouldn't need to think about a distributed algorithm in the first place. Still, in practice this is something to always check before rushing to the fancy solutions from this course!

2.5 BFS Tree Construction

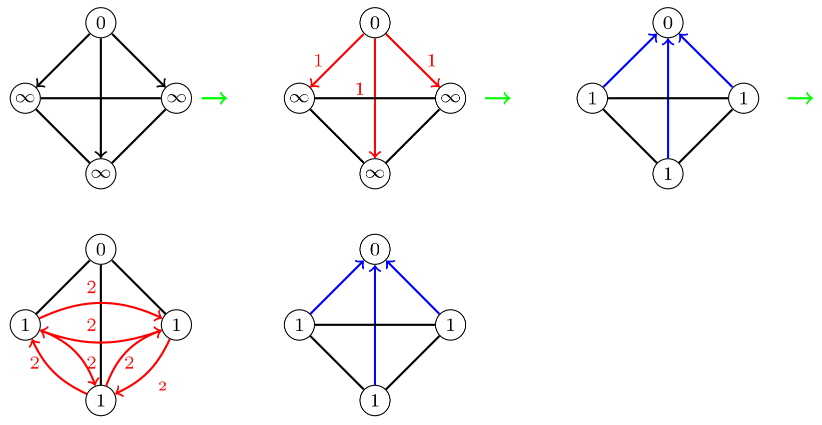

Figure 2.2: Trivial synchronous BFS tree construction. The root sends “1” to all neighbors. Nodes receiving a message “d” know they are in distance d from the root, send $d+1$ to all neighbors, and terminate. Here we see the algorithm in the complete graph; in general, it requires D synchronous rounds, where D is the diameter of the network.

Synchronizer β requires a tree to operate, preferably one with small depth d. In synchronous systems, flooding is a simple yet efficient method to construct a breadth-first search (BFS) spanning tree using at most $2|E|$ messages and ensuring that $d \leq D$, where

$$D := \max_{v,w\in V}\{\text{dist}(v,w)\}$$

is the network *diameter* and $\text{dist}(v,w)$ is the length of a shortest path from v

Algorithm 7 Dijkstra BFS

```
1: The algorithm proceeds in phases. In phase $i$ the nodes with distance $i$ to
   the root are detected. Let $T_i$ be the tree in phase $i$. We start with $T_0$ which
   is just the root.
2: repeat
3:   The root starts phase $i$ by broadcasting “start $i$” within $T_i$.
4:   When receiving “start i” a leaf node $v$ of $T_i$ (that is, a node that was
     newly discovered in the last phase) sends a “join $i+1$” message to all
     quiet neighbors. (A neighbor $w$ is quiet if $v$ has not yet “talked” to $w$.)
5:   A node $v$ receiving the first “join $i+1$” message replies with “ACK” and
     becomes a leaf of the tree $T_{i+1}$.
6:   A node $v$ receiving any further “join” message replies with “NACK”.
7:   The leaves of $T_i$ collect all the answers of their neighbors; then the leaves
     start an echo algorithm back to the root.
8:   When the echo process terminates at the root, the root increments the
     phase.
9: until there was no new node detected
```

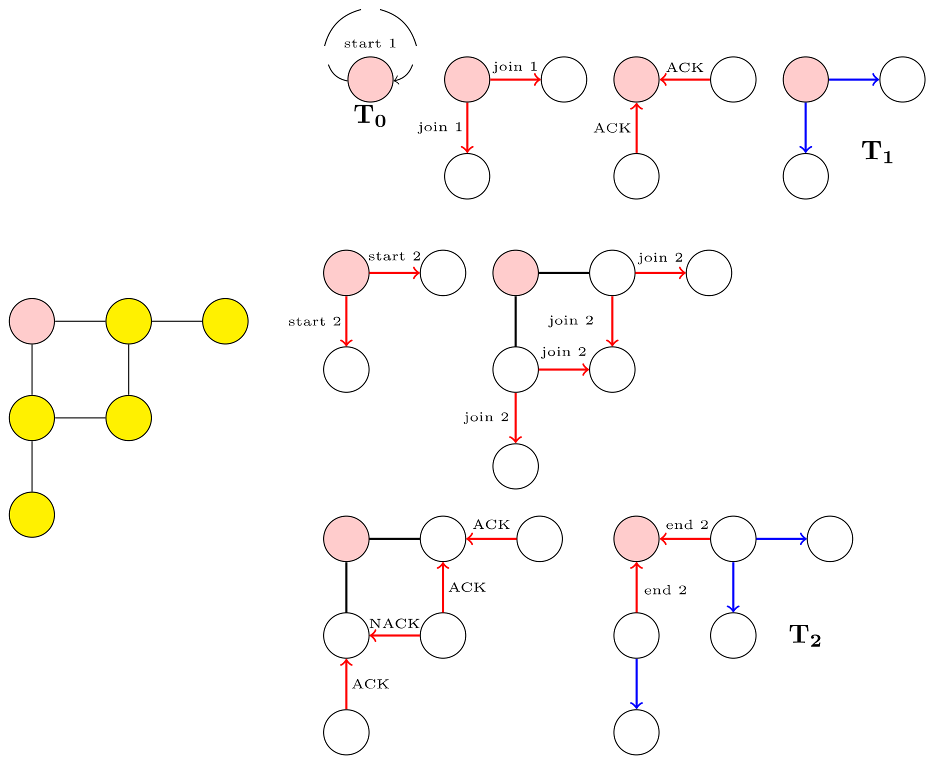

Figure 2.3: A run of Dijkstra's algorithm on the graph depicted on the left. In the first phase, the neighbors of the root join the tree, resulting in $T1$. The second phase uses the existing tree edges to communicate it's start and finish, very similar to the β-synchronizer. If the network had a larger diameter than 2, there would be more phases following the same pattern.

to w. However, in asynchronous systems the spanning tree constructed by the flooding algorithm may be far from BFS; in the worst case, we construct a line in a complete graph (see Figure 2.4)!

In this section, we implement two classic BFS constructions—Dijkstra and Bellman-Ford—as asynchronous algorithms. We start with the Dijkstra algorithm. The basic idea is to always add the "closest" node to the existing part of the BFS tree. We parallelize this idea by developing the BFS tree layer by layer. In Algorithm 7, by "broadcast" we denote the operation that the root uses flooding to distribute some information throughout the (current). "Echo" means the process of all leaves sending some information to the root, which interior nodes aggregate from all children before forwarding it.

Theorem 2.11 (Distributed Dijkstra)**.** *The time complexity of Algorithm 7 is $\mathcal{O}(D^2)$. Its message complexity is $\mathcal{O}(|E| + nD)$.*

Proof. A broadcast/echo algorithm in T_p needs at most time $2D$. Finding new neighbors at the leaves costs 2 time units. Since the depth of the BFS tree is bounded by the diameter, we have D phases, giving a total time complexity of

$\mathcal{O}(D^2)$.

The broadcast/echo routine uses each edge of a tree twice, i.e., at most $2(n-1)$ such messages are sent in each phase. Since there are D phases, this amounts to $\mathcal{O}(nD)$ messages. On each edge, there are at most 2 "join" messages. Replies to a "join" request are answered by 1 "ACK" or "NACK", which means that we have at most 4 additional messages per edge. Therefore the message complexity is $\mathcal{O}(|E| + nD)$. □

Remarks:

- The description of the algorithm is less formal than before, but highlights the structure of the algorithm better. This helps with explaining the idea, but don't be fooled: this style can easily trick the reader (or writer!) into believing some things will happen in a certain order, while in fact asynchrony could cause something entirely different to happen!
- We haven't specified how the root is selected. Either one has to specify this in advance, or the problem of *leader election* has to be solved. This is another fundamental problem in distributed computing.

The basic idea of Bellman-Ford is even simpler. We keep book on the distance to the root. If a neighbor has found a better route to the root, a node might also need to update its distance.

Algorithm 8 Bellman-Ford BFS

```
1: Each node $v$ stores an integer $d_v$ which corresponds to the distance $\mathrm{dist}(v, v_0)$
   to the root $v_0$. Initially $d_{v_0} = 0$, and $d_v = \infty$ for $v \in V \setminus \{v_0\}$.
2: The root starts the algorithm by sending "1" to all neighbors.
3: if $v$ receives "$d$" with $d < d_v$ from $w$ then
4:    $d_v := d$
5:    $p_v := w$
6:    $v$ sends "$d + 1$" to all neighbors
7: end if
```

Theorem 2.12 (Bellman-Ford BFS)**.** *The time complexity of Algorithm 8 is $\mathcal{O}(D)$, the message complexity is $\mathcal{O}(n|E|)$.*

Proof. We prove the time complexity by induction. We claim that a node at distance d from the root has received a message "d" by time d. The root knows by time 0 that it is the root. A node v at distance d has a neighbor w at distance $d-1$. Node w by induction sends a message "d" to v by time $d-1$, which is then received by v by time d.

Regarding message complexity, a node can reduce its distance at most $n-1$ times; each of these times it sends a message to all its neighbors. If all nodes do this we have $\mathcal{O}(n|E|)$ messages. □

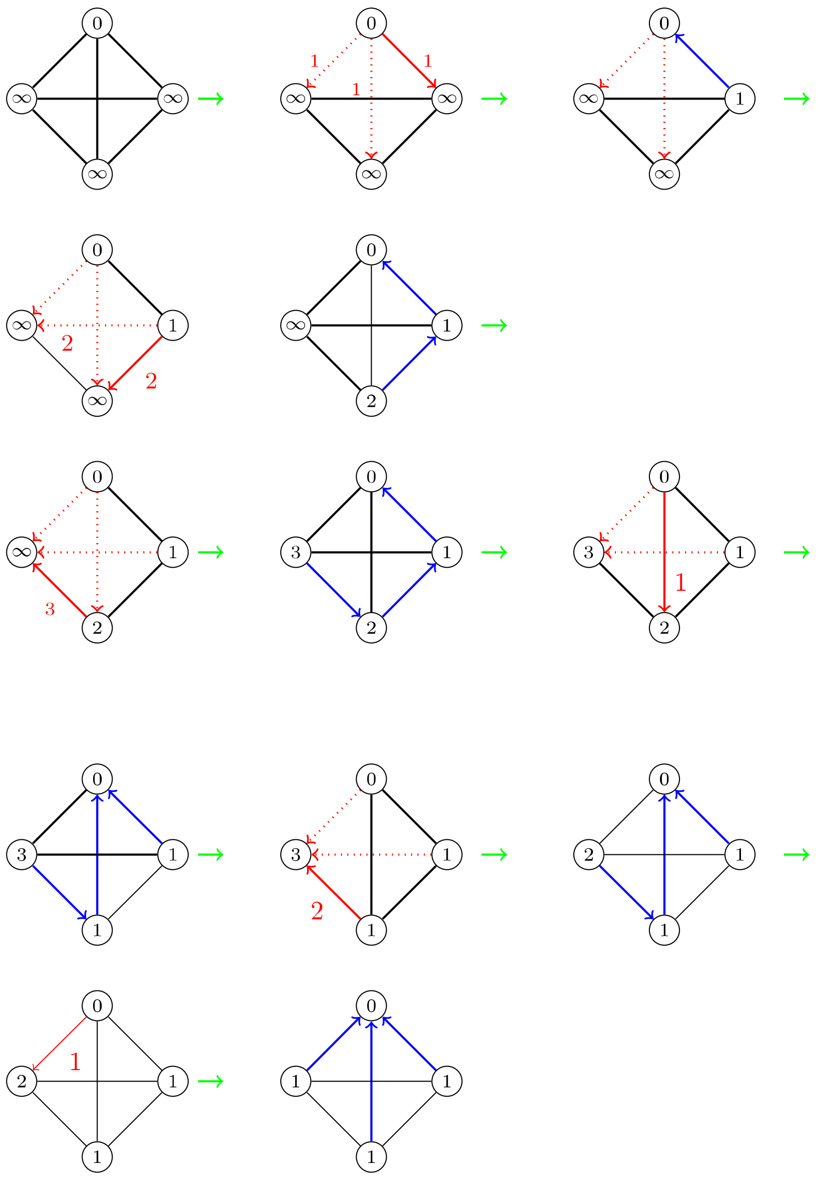

Figure 2.4: “Bad” execution of the Bellman-Ford algorithm on the complete graph of 4 nodes. Red arrows indicate messages that cause nodes’ estimated distances to change upon reception. Dotted arrows indicate that such messages are delayed. Note that terminating when the first distance estimate is obtained (i.e., running a naive synchronous BFS algorithm) would yield erroneous results. Moreover, on average each node changes its label $n/2$ times (here $n = 4$), implying a message complexity of $n \cdot n \cdot n/2 \in \Theta(n^3)$.

Remarks:

- Here’s another “wrong” algorithm/proof: I didn’t say in the algorithm or the theorem how termination is detected. Can you see how to fix this?

- If not, don't worry. We'll cover this in the exercises, with astonishing results!
- Algorithm 7 has the better message complexity and Algorithm 8 has the better time complexity. The currently best algorithm (optimizing both) needs $\mathcal{O}(|E| + n \log^3 n)$ messages and $\mathcal{O}(D \log^3 n)$ time. This "trade-off" algorithm is beyond the scope of this lecture.
- As such an advanced algorithm is quite efficient and one usually can construct the tree in advance, in many cases it is not a big deal. Still, one needs to keep in mind that this initial overhead exists and might not be worth it.

2.6 Hybrid Synchronizers

Synchronizer α is fast, but costs a lot of messages. Synchronizer β is slow, but efficient in terms of messages. Can we compromise? The answer is yes and called (surprise!) synchronizer γ.

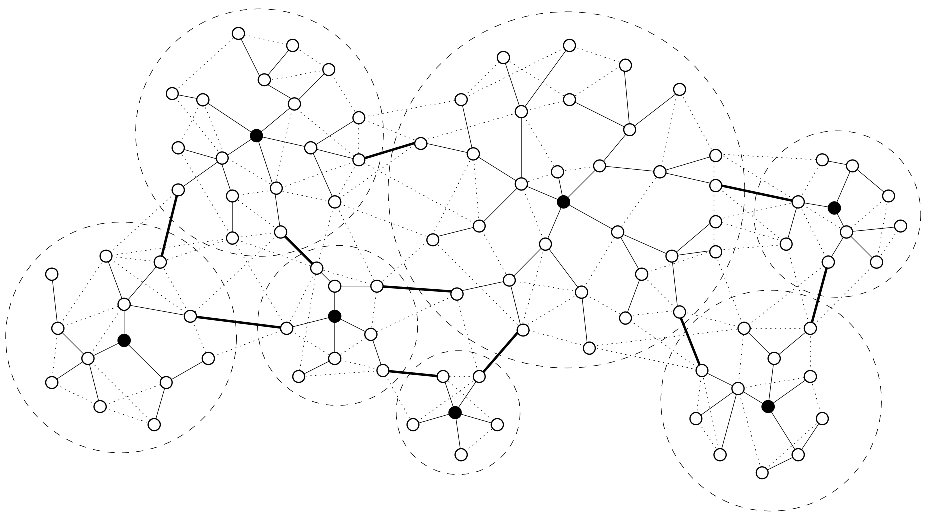

Figure 2.5: A cluster partition of a network: The dashed cycles specify the clusters, cluster leaders are black, the solid edges are the edges of the intracluster trees, and the bold solid edges are the intercluster edges.

We will now briefly discuss the key ideas (there has been enough detail for one lecture already!). In the initialization phase, the network is partitioned into clusters of small diameter. In each cluster, a leader node is chosen and a BFS tree rooted at this leader node is computed. These trees are called the *intracluster trees.* Two clusters C_1 and C_2 are called neighboring if there are nodes $u \in C_1$ and $v \in C_2$ for which $(u, v) \in E$. For every two neighboring clusters, an *intercluster edge* is chosen, which will serve for communication between these clusters. Figure 2.5 illustrates this partitioning into clusters.

We say that a cluster is safe if all its nodes are safe. Let's start the description from all nodes generating a pulse (which, of course, may happen at very different times). In at most 2 time units, all nodes will be safe. As in synchronizer β, we now let the leader of each cluster learn that the cluster is same. Then, the leader will let the leaders of adjacent clusters know that its cluster is safe. This is done by flooding this information through the own cluster using the tree, then communicating it over the intercluster edges, and using the same approach as in the β-synchronizer to collect the information that all adjacent clusters are safe within the cluster (i.e., a "safe" message is sent to the parent once "safe" via all incident intercluster edges and from all children has been received). Once a cluster leader knows that all adjacent clusters and its own are safe, it tells all nodes in its cluster to generate the next pulse. Hence, we essentially apply synchronizer α between clusters.

Theorem 2.13 (Synchronizer γ)**.** *Let E_C be the set of intercluster edges and let k be the maximum cluster radius (i.e., the maximum distance of a leaf to its cluster leader). Simulating a synchronous T-round algorithm sending M messages using synchronizer γ then takes $\mathcal{O}((T+1)k)$ time and requires $\mathcal{O}(M+(T+1)(|E_C|+n))$ messages.*

Proof sketch. From the above description, we see that each pulse requires a constant number of flooding or echo operations on the trees (which have depth at most k and in total at most $n-1$ edges), 2 messages over each intercluster edge, and the messages of the simulated algorithm plus acknowledgements. Hence, each pulse takes k time, $\mathcal{O}(|E_C|+n)$ synchronizer messages, and two times the number of messages sent by the algorithm in the simulated round. Summing up, over T rounds (and taking into account handling of termination), we get the stated bounds. □

In the exercises, we will see that one can have $|E_C| \in \mathcal{O}(n^{1+1/k})$, which is pretty impressive.

Corollary 2.14 (Synchronizer γ)**.** *For $k \in \{1, \dots, \lceil \log n \rceil\}$, there is a network partion so that synchronizter γ simulates a synchronous T-round algorithm sending M messages in $\mathcal{O}((T+1)k)$ time and requires $\mathcal{O}(M+(T+1)n^{1+1/k})$ messages.*

Remarks:

- For $k = 1$, this is just like synchronizer α.
- For $k = \lceil \log n \rceil$, this uses as few messages as synchronizer β, but pays only a factor of $\log n$ in time, regardless of D!

Remarks:

- It can be shown that the trade-off between cluster radius and number of intercluster edges of Corollary 2.14 is asymptotically optimal. There are graphs for which every clustering into clusters of radius at most k requires $n^{1+c/k}$ intercluster edges for some constant c.

- The synchronizers β and γ achieve global synchronization, i.e. every node generates every clock pulse. The disadvantage of this is that nodes that do not participate in a computation also have to participate in the synchronization. In many computations (e.g. in a BFS construction), many nodes only participate for a few synchronous rounds. In such scenarios, it is possible to achieve time and message complexity $\mathcal{O}(\log^3 n)$ per synchronous round (without initialization).
- It can be shown that if all nodes in the network need to generate all pulses, the trade-off of synchronizer γ is asymptotically optimal.
- Partitions of networks into clusters of small diameter and coverings of networks with clusters of small diameters come in many variations and have various applications in distributed computations. In particular, apart from synchronizers, algorithms for routing, the construction of sparse spanning subgraphs, distributed data structures, and even computations of local structures such as a maximal independent sets are based on some kind of network partitions or covers.

What to take home

- Asynchrony does not affect solvability of problems – if there are no faults.
- It comes at a cost in time and/or message complexity, though.
- Simulation is a powerful tool for designing algorithms. Designing and analyzing advanced asynchronous algorithms can be very challenging. If it's ok to run a synchronizer (which simulates synchrony), things can become astronomically simpler.
- Not all problems have a single best solution. Frequently, there is a trade-off between quality measures that cannot be compared in a straightforward way, e.g. time vs. messages.
- Then again, there might be a "sweet spot", where the cost in either measure is very small. We had this with time vs. number of colors for list coloring, and with the γ-synchronizer using a good network partition.
- On the way to a good solution, it may turn out that one needs to solve another problem that did not appear to be of relevance initially.
- Totally unrelated: network partitions rock!

Chapter Notes

The idea behind synchronizers is quite intuitive and as such, synchronizers α and β were implicitly used in various asynchronous algorithms [Gal76, Cha79, CL85] before being proposed as separate entities. The general idea of applying synchronizers to run synchronous algorithms in asynchronous networks was first introduced by Awerbuch [Awe85]. His work also formally introduced the synchronizers α, β, and γ. Improved synchronizers that exploit inactive nodes or hypercube networks were presented in [AP90, PU87].

Trees are one of the oldest graph structures, already appearing in the first book about graph theory [Koe36]. *Broadcasting* (i.e., flooding some information through the network) in distributed computing is younger, but not that much [DM78]. Overviews about broadcasting can be found for example in Chapter 3 of [Pel00] and Chapter 7 of [HKP$^+$05]. Overviews of distributed tree construction can be found in Chapter 5 of [Pel00] or Chapter 4 of [Lyn96]. The classic papers on routing are [For56, Bel58, Dij59].

Figure 2.5 is courtesy of Roger Wattenhofer. Wide parts of this lecture are based on the corresponding lecture of his course "Principles of Distributed Computing".

Bibliography

[AP90] Baruch Awerbuch and David Peleg. Network Synchronization with Polylogarithmic Overhead. In *Proceedings of the 31st IEEE Symposium on Foundations of Computer Science (FOCS)*, 1990.

[Awe85] Baruch Awerbuch. Complexity of Network Synchronization. *Journal of the ACM (JACM)*, 32(4):804–823, October 1985.

[Bel58] Richard Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[Cha79] E.J.H. Chang. *Decentralized Algorithms in Distributed Systems.* PhD thesis, University of Toronto, 1979.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 1:63–75, 1985.

[Dij59] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[DM78] Y.K. Dalal and R.M. Metcalfe. Reverse path forwarding of broadcast packets. *Communications of the ACM*, 12:1040–148, 1978.

[For56] Lester R. Ford. Network Flow Theory. *The RAND Corporation Paper P-923*, 1956.

[Gal76] Robert Gallager. Distributed Minimum Hop Algorithms. Technical report, Lab. for Information and Decision Systems, 1976.

[HKP$^+$05] Juraj Hromkovic, Ralf Klasing, Andrzej Pelc, Peter Ruzicka, and Walter Unger. *Dissemination of Information in Communication Networks - Broadcasting, Gossiping, Leader Election, and Fault-Tolerance.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2005.

[Koe36] Denes Koenig. *Theorie der endlichen und unendlichen Graphen.* Teubner, Leipzig, 1936.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[Pel00] David Peleg. *Distributed computing: a locality-sensitive approach.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[PU87] David Peleg and Jeffrey D. Ullman. An Optimal Synchronizer for the Hypercube. In *Proceedings of the sixth annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 77–85, 1987.