complexity of the resulting asynchronous algorithm depends on the overhead introduced by the synchronizer. For a synchronizer $S$, let $T(S)$ and $M(S)$ be the time and message complexities of $S$ for each generated clock pulse. As we will see, some of the synchronizers need an initialization phase. We denote the time and message complexities of the initialization by $T(S)$ and $M(S)$, respectively. If $T(A)$ and $M(A)$ are the time and message complexities of the given synchronous algorithm $A$, the total time and message complexities $T_{\text{tot}}$ and $M_{\text{tot}}$ of the resulting asynchronous algorithm then become

$$T_{\text{tot}} = T(S) + T(A) \cdot (1 + T(S)) \quad \text{and} \quad M_{\text{tot}} = M(S) + M(A) + T(A) \cdot M(S),$$

respectively.

Remarks:

- Because the initialization only needs to be done once for each network, we will mostly be interested in the overheads $T(S)$ and $M(S)$ per round of the synchronous algorithm.

Definition 10.3 (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.

Lemma 10.4. If all neighbors of a node $v$ are safe with respect to the current clock pulse of $v$, the next pulse can be generated for $v$.

Proof. If all neighbors of $v$ are safe with respect to a certain pulse, $v$ has received all messages of the given pulse. Node $v$ therefore satisfies the condition of Definition 10.1 for generating a valid next pulse. \hfill $\square$

Remarks:

- In order to detect safety, we require that all algorithms send acknowledgments for all received messages. As soon as a node $v$ has received an acknowledgement for each message that it has sent in a certain pulse, it knows that it is safe with respect to that pulse. Note that sending acknowledgments does not increase the asymptotic time and message complexities.

10.2 The Local Synchronizer $\alpha$

Algorithm 40 Synchronizer $\alpha$ (at node $v$)

1. wait until $v$ is safe
2. send SAFE to all neighbors
3. wait until $v$ receives SAFE messages from all neighbors
4. start new pulse

Synchronizer $\alpha$ is very simple. It does not need an initialization. Using acknowledgments, each node eventually detects that it is safe. It then reports this fact directly to all its neighbors. Whenever a node learns that all its neighbors are safe, a new pulse is generated. Algorithm 40 formally describes the synchronizer $\alpha$. 

101

CHAPTER 10. SYNCHRONIZATION
10.3. SYNCHRONIZER $\beta$

Theorem 10.5. The time and message complexities of synchronizer $\alpha$ per synchronous round are

\[ T(\alpha) = O(1) \quad \text{and} \quad M(\alpha) = O(m). \]

Proof. Communication is only between neighbors. As soon as all neighbors of a node $v$ become safe, $v$ knows of this fact after one additional time unit. For every clock pulse, synchronizer $\alpha$ sends at most four additional messages over every edge. Each of the nodes may have to acknowledge a message and report safety.

Remarks:

- Synchronizer $\alpha$ was presented in a framework, mostly set up to have a common standard to discuss different synchronizers. Without the framework, synchronizer $\alpha$ can be explained more easily:
  1. Send message to all neighbors, include round information $i$ and actual data of round $i$ (if any).
  2. Wait for message of round $i$ from all neighbors, and go to next round.
- Although synchronizer $\alpha$ allows for simple and fast synchronization, it produces awfully many messages. Can we do better? Yes.

10.3 The Global Synchronizer $\beta$

Algorithm 41 Synchronizer $\beta$ (at node $v$)

1. wait until $v$ is safe
2. wait until $v$ receives SAFE messages from all its children in $T$
3. if $v \neq \ell$ then
   4. send SAFE message to parent in $T$
   5. wait until PULSE message received from parent in $T$
   6. end if
7. send PULSE message to children in $T$
8. start new pulse

Synchronizer $\beta$ needs an initialization that computes a leader node $\ell$ and a spanning tree $T$ rooted at $\ell$. As soon as all nodes are safe, this information is propagated to $\ell$ by a convergecast. The leader then broadcasts this information to all nodes. The details of synchronizer $\beta$ are given in Algorithm 41.

Theorem 10.6. The time and message complexities of synchronizer $\beta$ per synchronous round are

\[ T(\beta) = O(\text{diameter}(T)) \leq O(n) \quad \text{and} \quad M(\beta) = O(n). \]

The time and message complexities for the initialization are

\[ T_{\text{init}}(\beta) = O(n) \quad \text{and} \quad M_{\text{init}}(\beta) = O(m + n \log n). \]

Remark: We now got a time-efficient synchronizer ($\beta$) and a message-efficient synchronizer ($\gamma$), it is only natural to ask whether we can have the best of both worlds. And, indeed, we can. How is that synchronizer called? Quite obviously: $\gamma$.

10.4 The Hybrid Synchronizer $\gamma$

![Figure 10.1: 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.](image)

Proof. Because the diameter of $T$ is at most $n - 1$, the convergecast and the broadcast together take at most $2n - 2$ time units. Per clock pulse, the synchronizer sends at most $2n - 2$ synchronization messages (one in each direction over each edge of $T$).

With an improvement (due to Awerbuch) of the GHS algorithm (Algorithm 15) you saw in Chapter 3, it is possible to construct an MST in time $O(n)$ with $O(m + n \log n)$ messages in an asynchronous environment. Once the tree is computed, the tree can be made rooted in time $O(n)$ with $O(n)$ messages.

Remarks:

- We now got a time-efficient synchronizer ($\gamma$) and a message-efficient synchronizer ($\beta$), it is only natural to ask whether we can have the best of both worlds. And, indeed, we can. How is that synchronizer called? Quite obviously: $\gamma$. 

Figure 10.1 illustrates this partitioning into clusters. We will discuss the details of how to construct such a partition in the next section. We say that a cluster is safe if all its nodes are safe.
10.4. SYNCHRONIZER \( \gamma \)

Synchronizer \( \gamma \) works in two phases. In a first phase, synchronizer \( \beta \) is applied separately in each cluster by using the intracluster trees. Whenever the leader of a cluster learns that its cluster is safe, it reports this fact to all the nodes in the clusters as well as to the leaders of the neighboring clusters. Now, the nodes of the cluster enter the second phase where they wait until all the neighboring clusters are known to be safe and then generate the next pulse. Hence, we essentially apply synchronizer \( \alpha \) between clusters. A detailed description is given by Algorithm 42.

Algorithm 42 Synchronizer \( \gamma \) (at node \( v \))

1. wait until \( v \) is safe
2. wait until \( v \) receives SAFE messages from all children in intracluster tree
3. if \( v \) is not cluster leader then
   4. send SAFE message to parent in intracluster tree
   5. wait until CLUSTERSAFE message received from parent
   end if
7. send CLUSTERSAFE message to all children in intracluster tree
8. send NEIGHBORSAFE message over all intercluster edges of \( v \)
9. wait until \( v \) receives NEIGHBORSAFE messages from all adjacent intercluster edges and all children in intracluster tree
10. if \( v \) is not cluster leader then
   11. send NEIGHBORSAFE message to parent in intracluster tree
   12. wait until PULSE message received from parent
   end if
14. send PULSE message to children in intracluster tree
15. start new pulse

Theorem 10.7. Let \( m_C \) be the number of intercluster edges and let \( k \) be the maximum cluster radius (i.e., the maximum distance of a leaf to its cluster leader). The time and message complexities of synchronizer \( \gamma \) are

\[
T(\gamma) = O(k) \quad \text{and} \quad M(\gamma) = O(n + m_C).
\]

Proof. We ignore acknowledgements, as they do not affect the asymptotic complexities. Let us first look at the number of messages. Over every intracluster tree edge, exactly one SAFE message, one CLUSTERSAFE message, one NEIGHBORSAFE message, and one PULSE message is sent. Further, one NEIGHBORSAFE message is sent over every intercluster edge. Because there are less than \( n \) intracluster tree edges, the total message complexity therefore is at most \( 4m + 2m_C = O(n + m_C) \).

For the time complexity, note that the depth of each intracluster tree is at most \( k \). On each intracluster tree, two convergecasts (the SAFE and NEIGHBORSAFE messages) and two broadcasts (the CLUSTERSAFE and PULSE messages) are performed. The time complexity for this is at most \( 4k \). There is one more time unit needed to send the NEIGHBORSAFE messages over the intercluster edges. The total time complexity therefore is at most \( 4k + 1 = O(k) \).

10.5 Network Partition

We will now look at the initialization phase of synchronizer \( \gamma \). Algorithm 43, \( B(v,r) \) denotes the ball of radius \( r \) around \( v \), i.e., \( B(v,r) = \{ u \in V : d(u,v) \leq r \} \) where \( d(u,v) \) is the hop distance between \( u \) and \( v \). The algorithm has a parameter \( \rho > 1 \). The clusters are constructed sequentially. Each cluster is started at an arbitrary node that has not been included in a cluster. Then the cluster radius is grown as long as the cluster grows by a factor more than \( \rho \).

Algorithm 43 Cluster construction

1. while unprocessed nodes do
2. select an arbitrary unprocessed node \( v \);
3. \( r := 0 \);
4. while \( |B(v,r+1)| > \rho |B(v,r)| \) do
5. \( r := r + 1 \);
6. end while
7. makeCluster(\( B(v,r) \)) // all nodes in \( B(v,r) \) are now processed
8. end while

Remarks:
- The algorithm allows a trade-off between the cluster diameter \( k \) (and thus the time complexity) and the number of intercluster edges \( m_C \) (and thus the message complexity). We will quantify the possibilities in the next section.
- Two very simple partitions would be to make a cluster out of every single node or to make one big cluster that contains the whole graph. We then get synchronizers \( \alpha \) and \( \beta \) as special cases of synchronizer \( \gamma \).

Theorem 10.8. Algorithm 43 computes a partition of the network graph into clusters of radius at most \( \log_2 n \). The number of intercluster edges is at most \( (\rho - 1) n \).

Proof. The radius of a cluster is initially 0 and does only grow as long as \( n \) grows by a factor larger than \( \rho \). Since there are only \( n \) nodes in the graph, this can happen at most \( \log_2 n \) times.

To count the number of intercluster edges, observe that an edge can only become an intercluster edge if it connects a node at the boundary of a cluster with a node outside a cluster. Consider a cluster \( C \) of size \( |C| \). We know that \( C = B(v,r) \) for some \( v \in V \) and \( r \geq 0 \). Further, we know that \( |B(v,r+1)| \leq \rho |B(v,r)| \). The number of nodes adjacent to cluster \( C \) is therefore at most \( |B(v,r+1) \setminus B(v,r)| \leq \rho |C| - |C| \). Because there is only one intercluster edge connecting two clusters by definition, the number of intercluster edges adjacent to \( C \) is at most \( (\rho - 1) |C| \). Summing over all clusters, we get that the total number of intercluster edges is at most \( (\rho - 1) n \). \( \square \)

Corollary 10.9. Using \( \rho = 2 \), Algorithm 43 computes a clustering with cluster radius at most \( \log_2 n \) and with at most \( n \) intercluster edges.
Corollary 10.10. Using $r = n^{1/k}$, Algorithm 43 computes a clustering with cluster radius at most $k$ and at most $O(n^{1+1/k})$ intercluster edges.

Remarks:
- Algorithm 43 describes a centralized construction of the partitioning of the graph. For $r \geq 2$, the clustering can be computed by an asynchronous-distributed algorithm in time $O(n)$ with $O(m + n \log n)$ (reasonably sized) messages (showing this will be part of the exercises).
- It can be shown that the trade-off between cluster radius and number of intercluster edges of Algorithm 43 is asymptotically optimal. There are graphs for which every clustering into clusters of radius at most $k$ requires $n^{1+1/k}$ intercluster edges for some constant $c$.
- The above remarks lead to a complete characterization of the complexity of synchronizer $\gamma$.

Corollary 10.11. The time and message complexities of synchronizer $\gamma$ per synchronous round are

\[
T(\gamma) = O(k) \quad \text{and} \quad M(\gamma) = O(n^{1+1/k}).
\]

The time and message complexities for the initialization are

\[
T_{\text{init}}(\gamma) = O(n) \quad \text{and} \quad M_{\text{init}}(\gamma) = O(m + n \log n).
\]

Remarks:
- The synchronizer idea and the synchronizers discussed in this chapter are due to Baruch Awerbuch.
- In Chapter 3, you have seen that by using flooding, there is a very simple synchronous algorithm to compute a BFS tree in time $O(D)$ with message complexity $O(m)$.
- If we use synchronizer $\gamma$ to make this algorithm asynchronous, we get an algorithm with time complexity $O(n + D \log n)$ and message complexity $O(m + n \log n + D \cdot n)$ (including initialization).
- The synchronizers $\alpha$, $\beta$, and $\gamma$ achieve global synchronization, i.e., every node generates every clock pulse. The disadvantage of this is that nodes 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. An improved synchronizer due to Awerbuch and Peleg can exploit such a scenario and achieves time and message complexity $O(n \log^2 n)$ 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 $\gamma$ 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 MIS or a dominating set are based on some kind of network partitions or covers.

10.6 Clock Synchronization

“A man with one clock knows what time it is – a man with two is never sure.”

Synchronizers can directly be used to give nodes in an asynchronous network a common notion of time. In wireless networks, for instance, many basic protocols need an accurate time. Sometimes a common time in the whole network is needed, often it is enough to synchronize neighbors. The purpose of the time division multiple access (TDMA) protocol is to use the common wireless channel as efficiently as possible, i.e., interfering nodes should never transmit at the same time (on the same frequency). If we use synchronizer $\beta$ to give the nodes a common notion of time, every single clock cycle costs $D$ time units!

Often, each (wireless) node is equipped with an internal clock. Using this clock, it should be possible to divide time into slots, and make each node send (or listen, or sleep, respectively) in the appropriate slots according to the media access control (MAC) layer protocol used.

However, as it turns out, synchronizing clocks in a network is not trivial. As nodes’ internal clocks are not perfect, they will run at speeds that are time-dependent. For instance, variations in temperature or supply voltage will affect the time units!

As nodes’ internal clocks are not perfect, they will run at speeds that are time-dependent. For instance, variations in temperature or supply voltage will affect the time units!

The purpose of the time division multiple access (TDMA) protocol is to use the common wireless channel as efficiently as possible, i.e., interfering nodes should never transmit at the same time (on the same frequency). If we use synchronizer $\beta$ to give the nodes a common notion of time, every single clock cycle costs $D$ time units!

Often, each (wireless) node is equipped with an internal clock. Using this clock, it should be possible to divide time into slots, and make each node send (or listen, or sleep, respectively) in the appropriate slots according to the media access control (MAC) layer protocol used.
10.6. CLOCK SYNCHRONIZATION

where the message delivery time is always last for one node and slow for the other and the logical clocks are off by 1/2. In both scenarios, the messages sent at time $i$ according to the clock of the sender arrive at time $i + 1/2$ according to the logical clock of the receiver. Therefore, for nodes $u$ and $v$, both cases with clock drift seem the same as the case with perfectly synchronized clocks. Furthermore, in a linked list of $D$ nodes, the left- and rightmost node $l, r$ cannot distinguish $t_l = t_r + D/2$ from $t_l = t_r - D/2$.

Remarks:

- From Theorem 10.12, it directly follows that all the clock synchronization algorithms we studied have a global skew of $\Omega(D)$.
- Many natural algorithms manage to achieve a global clock skew of $O(D)$.

As both the message jitter and hardware clock drift are bounded by constants, it helps us to be able to get a constant drift between neighboring nodes. As synchronizer $\alpha$ pays most attention to the local synchronization, we take a look at a protocol inspired by the synchronizer $\alpha$. A pseudo-code representation for the clock synchronization protocol $\alpha$ is given in Algorithm 44.

Algorithm 44 Clock synchronization $\alpha$ (at node $v$)

1: repeat
2: send logical time $t$ to all neighbors
3: if Receive logical time $t_u$, where $t_u > t$, from any neighbor $u$ then
4: $t_u := t_u$
5: end if
6: until done

Lemma 10.13. The clock synchronization protocol $\alpha$ has a local skew of $\Omega(n)$.

Proof. Let the graph be a linked list of $D$ nodes. We denote the nodes by $v_1, v_2, \ldots, v_D$ from left to right and the logical clock of node $v_i$ by $t_i$. Apart from the left-most node $v_1$ all hardware clocks run with speed 1 (real time).

Node $v_1$ runs at maximum speed, i.e., the time between two pulses is not 1 but $1 - \epsilon$. Assume that initially all message delays are 1. After some time, node $v_1$ will start to speed up $v_2$ and after some more time $v_2$ will speed up $v_3$ and so on. At some point of time, we will have a clock skew of 1 between any two neighbors. In particular, $t_1 = T + D - 1$.

Now we start playing around with the message delays. Let $t_1 = T$. First we set the delay between the $v_1$ and $v_2$ to 0. Now node $v_2$ immediately adjusts its logical clock to $T$. After this event (which is instantaneous in our model) we set the delay between $v_2$ and $v_3$ to 0, which results in $v_3$ setting its logical clock to $T$ as well. We perform this successively to all pairs of nodes until $v_D-2$ and $v_D-1$. Now node $v_D$ sets its logical clock to $T$, which indicates that the difference between the logical clocks of $v_D-1$ and $v_D$ is $T - (T - (D - 1)) = D - 1$.

Remarks:

- The introduced examples may seem cooked-up, but examples like this exist in all networks, and for all algorithms. Indeed, it was shown that any natural clock synchronization algorithm must have a bad local skew. In particular, a protocol that averages between all neighbors is even worse than the introduced $\alpha$ algorithm. This algorithm has a clock skew of $\Omega(D)$ in the linked list, at all times.
- Recently, there was a lot of progress in this area, and it was shown that the local clock skew is $\Omega(\log D)$. Naturally, as synchronizers are motivated by practical difficulties with local clocks, there are plenty of real-life applications. Studies regarding applications can be found in, e.g., [SM86, Awe85b, LTC89, AP90, PU87]. Synchronizers in the presence of network failures have been discussed in [AP88, HS94].

Chapter Notes

The idea behind synchronizers is quite intuitive and as such, synchronizers $\alpha$ and $\beta$ were implicitly used in various asynchronous algorithms [GL76, 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 [Awe85a]. His work also formally introduced the synchronizers $\alpha$ and $\beta$, whereas other constructions were presented in [AP90, PU87].

Naturally, as synchronizers are motivated by practical difficulties with local clocks, there are plenty of real-life applications. Studies regarding applications can be found in, e.g., [SM86, Awe85b, LTC89, AP90, PU87]. Synchronizers in the presence of network failures have been discussed in [AP88, HS94].

It has been known for a long time that the global clock skew is $\Omega(D)$ [LL84, ST87]. The problem of synchronizing the clocks of nearby nodes was introduced by Fan and Lynch in [LF04]; they proved a surprising lower bound of $\Omega(\log D / \log \log D)$ for the local skew. The first algorithm providing a non-trivial local skew of $O(D^2)$ was given in [LLW06]. Later, matching upper and lower bounds of $\Omega(\log D)$ were given in [LLW10]. The problem has also been studied in a dynamic setting [KLO09, KLO10].

Clock synchronization is a well-studied problem in practice, for instance regarding the global clock skew in sensor networks, e.g., [EG02, GKS03, MRSL04, PSJ04]. One more recent line of work is focusing on the problem of minimizing the local clock skew [BvRW07, SW09, LSW09, FW10, FZTS11].

Bibliography


Fabian Kuhn, Thomas Locher, and Rotem Oshman. Gradient Clock Synchronization in Dynamic Networks. In 29th Symposium on Principles of Distributed Computing (PODC), Zurich, Switzerland, July 2010.


