

COMP9243 — Week 5 (12s1)

Ihor Kuz, Manuel M. T. Chakravarty & Gernot Heiser

Synchronisation & Coordination

This lecture deals with one of the fundamental issues encountered when constructing a system made up of independent communicating processes: dealing with time and making sure that processes do the right thing at the right time. In essence this comes down to allowing processes to synchronise and coordinate their actions. Coordination refers to coordinating the actions of separate processes relative to each other and allowing them to agree on global state (such as values of a shared variable). Synchronisation is coordination with respect to time, and refers to the ordering of events and execution of instructions in time. Examples of synchronisation include ordering distributed events in a log file and ensuring that a process performs an action at a particular time. Examples of coordination include ensuring that processes agree on what actions will be performed (e.g., money will be withdrawn from the account), who will be performing actions (e.g., which replica will process a request), and the state of the system (e.g., the elevator is stopped).

Synchronisation and coordination play an important role in most distributed algorithms (i.e., algorithms intended to work in a distributed environment). In particular, some distributed algorithms are used to achieve synchronisation and coordination, while others assume the presence of synchronisation or coordination mechanisms. Discussions of distributed algorithms generally assume one of two timing models for distributed systems. The first is a synchronous model, where the time to perform all actions, communication delay, and clock drift on all nodes, are bounded. In asynchronous distributed systems there are no such bounds. Most real distributed systems are asynchronous, however, it is easier to design distributed algorithms for synchronous distributed systems. Algorithms for asynchronous systems are always valid on synchronous systems, however, the converse is not true.

Time & Clocks

As mentioned, time is an important concept when dealing with synchronisation and coordination. In particular it is often important to know when events occurred and in what order they occurred. In a nondistributed system dealing with time is trivial as there is a single shared clock. All processes see the same time. In a distributed system, on the other hand, each computer has its own clock. Because no clock is perfect each of these clocks has its own skew which causes clocks on different computers to drift and eventually become out of sync.

There are several notions of time that are relevant in a distributed system. First of all, internally a computer clock simply keeps track of ticks that can be translated into physical time (hours, minutes, seconds, etc.). This *physical time* can be global or local. Global time is a universal time that is the same for everyone and is generally based on some form of absolute time.¹ Currently *Coordinated Universal Time* (UTC), which is based on oscillations of the Cesium-133 atom, is the most accurate global time. Besides global time, processes can also consider local time. In this case the time is only relevant to the processes taking part in the distributed system (or algorithm). This time may be based on physical or logical clocks (which we will discuss later).

¹Although Einstein's special relativity theory shows that time is relative and there is, therefore, no absolute time, for our purposes (and at the worldwide scale) we can safely assume that such an absolute time does exist.

Physical Clocks

Physical clocks keep track of physical time. In distributed systems that rely on actual time it is necessary to keep individual computer clocks synchronised. The clocks can be synchronised to global time (*external synchronisation*), or to each other (*internal synchronisation*). Cristian's algorithm and the Network Time Protocol (NTP) are examples of algorithms developed to synchronise clocks to an external global time source (usually UTC). The Berkeley Algorithm is an example of an algorithm that allows clocks to be synchronised internally.

Cristian's algorithm requires clients to periodically synchronise with a central time server (typically a server with a UTC receiver). One of the problems encountered when synchronising clocks in a distributed system is that unpredictable communication latencies can affect the synchronisation. For example, when a client requests the current time from the time server, by the time the server's reply reaches the client the time will have changed. The client must, therefore, determine what the communication latency was and adjust the server's response accordingly. Cristian's algorithm deals with this problem by attempting to calculate the communication delay based on the time elapsed between sending a request and receiving a reply.

The Network Time Protocol is similar to Cristian's algorithm in that synchronisation is also performed using time servers and an attempt is made to correct for communication latencies. Unlike Cristian's algorithm, however, NTP is not centralised and is designed to work on a wide-area scale. As such, the calculation of delay is somewhat more complicated. Furthermore, NTP provides a hierarchy of time servers, with only the top layer containing UTC clocks. The NTP algorithm allows client-server and peer-to-peer (mostly between time servers) synchronisation. It also allows clients and servers to determine the most reliable servers to synchronise with. NTP typically provides accuracies between 1 and 50 msec depending on whether communication is over a LAN or WAN.

Unlike the previous two algorithms, the Berkeley algorithm does not synchronise to a global time. Instead, in this algorithm, a time server polls the clients to determine the average of everyone's time. The server then instructs all clients to set their clocks to this new average time. Note that in all the above algorithms a clock should never be set backward. If time needs to be adjusted backward, clocks are simply slowed down until time 'catches up'.

Logical Clocks

For many applications, the relative ordering of events is more important than actual physical time. In a single process the ordering of events (e.g., state changes) is trivial. In a distributed system, however, besides local ordering of events, all processes must also agree on ordering of causally related events (e.g., sending and receiving of a single message). Given a system consisting of N processes p_i , $i \in \{1, \dots, N\}$, we define the *local event ordering* \rightarrow_i as a binary relation, such that, if p_i observes e before e' , we have $e \rightarrow_i e'$. Based on this local ordering, we define a *global ordering* as a *happened before* relation \rightarrow , as proposed by Lamport [Lam78]: The relation \rightarrow is the smallest relation, such that

1. $e \rightarrow_i e'$ implies $e \rightarrow e'$,
2. for every message m , $send(m) \rightarrow receive(m)$, and
3. $e \rightarrow e'$ and $e' \rightarrow e''$ implies $e \rightarrow e''$ (transitivity).

The relation \rightarrow is almost a partial order (it lacks reflexivity). If $a \rightarrow b$, then we say a *causally affects* b . We consider unordered events to be *concurrent* if they are unordered; i.e.,

$$a \not\rightarrow b \text{ and } b \not\rightarrow a \text{ implies } a \parallel b.$$

As an example, consider Figure 1. We have the following causal relations:

$$\begin{aligned} E_{11} &\rightarrow E_{12}, E_{13}, E_{14}, E_{23}, E_{24}, \dots \\ E_{21} &\rightarrow E_{22}, E_{23}, E_{24}, E_{13}, E_{14}, \dots \end{aligned}$$

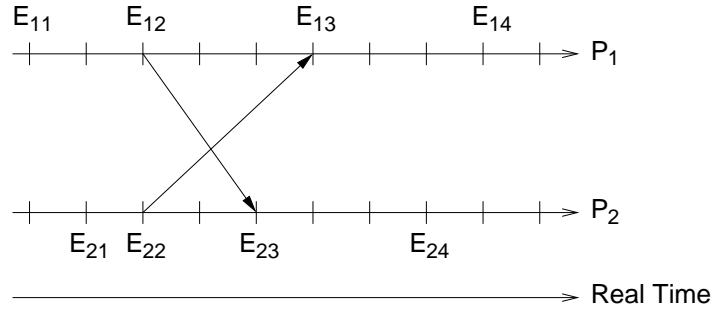


Figure 1: Example of event ordering

Moreover, the following events are concurrent: $E_{11} \parallel E_{21}$, $E_{12} \parallel E_{22}$, $E_{13} \parallel E_{23}$, $E_{11} \parallel E_{22}$, $E_{13} \parallel E_{24}$, $E_{14} \parallel E_{23}$, and so on.

Lamport Clocks

Lamport's logical clocks can be implemented as a software counter that locally computes the happened-before relation \rightarrow . This means that each process p_i maintains a *logical clock* L_i . Given such a clock, $L_i(e)$ denotes a Lamport timestamp of event e at p_i and $L(e)$ denotes a timestamp of event e at the process it occurred at. Processes now proceed as follows:

1. Before time stamping a local event, a process p_i executes $L_i := L_i + 1$.
2. Whenever a message m is sent from p_i to p_j :
 - Process p_i executes $L_i := L_i + 1$ and sends the new L_i with m .
 - Process p_j receives L_i with m and executes $L_j := \max(L_j, L_i) + 1$. $receive(m)$ is annotated with the new L_j .

In this scheme, $a \rightarrow b$ implies $L(a) < L(b)$, but $L(a) < L(b)$ does not necessarily imply $a \rightarrow b$. As an example, consider Figure 2. In this figure $E_{12} \rightarrow E_{23}$ and $L_1(E_{12}) < L_2(E_{23})$ (i.e., $2 < 3$), however we also have $E_{13} \not\rightarrow E_{24}$ while $L_1(E_{13}) < L_2(E_{24})$ (i.e., $3 < 4$).

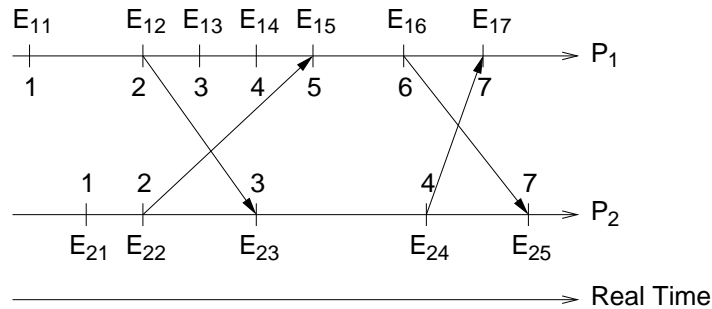


Figure 2: Example of the use of a Lamport's clocks

In some situations (e.g., to implement distributed locks), a partial ordering on events is not sufficient and a total ordering is required. In these cases, the partial ordering can be completed to total ordering by including process identifiers. Given local time stamps $L_i(e)$ and $L_j(e')$, we define global time stamps $\langle L_i(e), i \rangle$ and $\langle L_j(e'), j \rangle$. We, then, use standard lexicographical ordering, where $\langle L_i(e), i \rangle < \langle L_j(e'), j \rangle$ iff $L_i(e) < L_j(e')$, or $L_i(e) = L_j(e')$ and $i < j$.

Vector Clocks

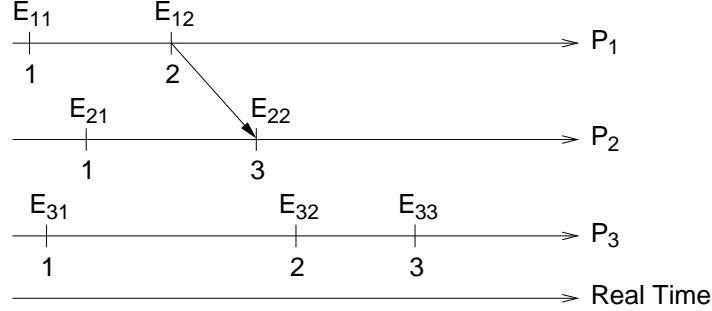


Figure 3: Example of the lack of causality with Lamport's clocks

The main shortcoming of Lamport's clocks is that $L(a) < L(b)$ does not imply $a \rightarrow b$; hence, we cannot deduce causal dependencies from time stamps. For example, in Figure 3, we have $L_1(E_{11}) < L_3(E_{33})$, but $E_{11} \not\rightarrow E_{33}$. The root of the problem is that clocks advance independently or via messages, but there is no history as to where advance comes from.

This problem can be solved by moving from scalar clocks to vector clocks, where each process maintains a vector clock V_i . V_i is a vector of size N , where N is the number of processes. The component $V_i[j]$ contains the process p_i 's knowledge about p_j 's clock. Initially, we have $V_i[j] := 0$ for $i, j \in \{1, \dots, N\}$. Clocks are advanced as follows:

1. Before p_i timestamps an event, it executes $V_i[i] := V_i[i] + 1$.
2. Whenever a message m is sent from p_i to p_j :
 - Process p_i executes $V_i[i] := V_i[i] + 1$ and sends V_i with m .
 - Process p_j receives V_i with m and *merges* the vector clocks V_i and V_j as follows:

$$V_j[k] := \begin{cases} \max(V_j[k], V_i[k]) + 1 & , \text{if } j = k \text{ (as in scalar clocks)} \\ \max(V_j[k], V_i[k]) & , \text{otherwise.} \end{cases}$$

This last part ensures that everything that subsequently happens at p_j is now causally related to everything that previously happened at p_i .

Under this scheme, we have, for all i, j , $V_i[i] \geq V_j[i]$ (i.e., p_i always has the most up-to-date version of its own clock); moreover, $a \rightarrow b$ iff $V(a) < V(b)$, where

- $V = V'$ iff $V[i] = V'[i]$ for all $i \in \{1, \dots, N\}$,
- $V \geq V'$ iff $V[i] \geq V'[i]$ for all $i \in \{1, \dots, N\}$,
- $V > V'$ iff $V \geq V' \wedge V \neq V'$; and
- $V \parallel V'$ iff $V \not\geq V' \wedge V' \not\geq V$

For example, consider the annotations at the diagram in Figure 4. Each event is annotated with both its vector clock value (the triple) and the corresponding value of a scalar Lamport clock. For $L_1(E_{12})$ and $L_3(E_{32})$, we have $2 = 2$ versus $(2, 0, 0) \neq (0, 0, 2)$. Likewise we have $L_2(E_{24}) > L_3(E_{32})$ but $(2, 4, 1) \not\geq (0, 0, 2)$ and thus $E_{32} \not\rightarrow E_{24}$.

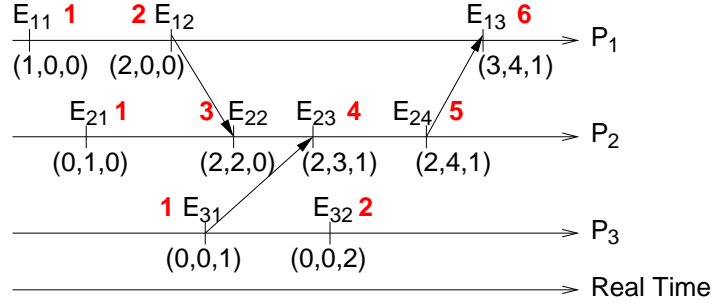


Figure 4: Example contrasting vector and scalar clock annotations

Global State

Determining global properties in a distributed system is often difficult, but crucial for some applications. For example, in distributed garbage collection, we need to be able to determine for some object whether it is referenced by any other objects in the system. Deadlock detection requires detection of cycles of processes infinitely waiting for each other. To detect the termination of a distributed algorithm we need to obtain simultaneous knowledge of all involved process as well as take account of messages that may still traverse the network. In other words, it is not sufficient to check the activity of all processes. Even if all processes appear to be passive, there may be messages in transition that, upon arrival, trigger further activity.

In the following, we are concerned with determining *stable* global states or properties that, once they occur, will not disappear without outside intervention. For example, once an object is no longer referenced by any other object (i.e., it may be garbage collected), no reference to the object can appear at a later time.

Consistent Cuts

To reason about the validity of global observations—i.e., observations that combine information from multiple nodes—the notion of *consistent cuts* is useful. Due to the lack of global time, we cannot simply require that all local observations must happen at the same time. As it is clear that using the state of the individual processes at arbitrary points in time is not generally going to result in a consistent overall picture, we need to define a criterion for determining when we regard a collection of local states to be globally consistent.

To formalise the notion of a consistent cut, we again refer to a system of N processes p_i , $i \in \{1, \dots, N\}$. Each process p_i , over time, proceeds through a series events $\langle e_i^0, e_i^1, e_i^2, \dots \rangle$, which we call p_i 's *history* denoted by h_i . This series may be finite or infinite. In any case, we denote by h_i^k a *k-prefix* of h_i (history of p_i up to and including event e_i^k). Each event e_i^j , as before, is either a local event or a communication event (e.g., sending or receiving of a message).

We denote the state of of any process p_i , immediately before event e_i^k , as s_i^k ; i.e., the state recording all events included in the history h_i^{k-1} . This makes s_i^0 refer to the initial state of p_i .

Using a total event ordering, we can merge all local histories into a *global history*

$$H = \bigcup_{i=1}^N h_i$$

and, similarly, we can combine a set of local states s_1, \dots, s_N into a global state $S = (s_1, \dots, s_N)$. This raises the question as to which combination of local states is *consistent* (a global state is consistent if for any received message in the state the corresponding send is also in the state). To answer this question, we need one more concept, namely that of a *cut*. Similar to the global

history, we can define cuts based on k -prefixes:

$$C = \bigcup_{i=1}^N h_i^{c_i}$$

where $h_i^{c_i}$ is history of p_i up to and including event $e_i^{c_i}$. The cut C corresponds to the state $S = (s_1^{c_1+1}, \dots, s_N^{c_N+1})$. The final events in a cut are its *frontier* defined as $\{e_i^{c_i} \mid i \in \{1, \dots, N\}\}$.

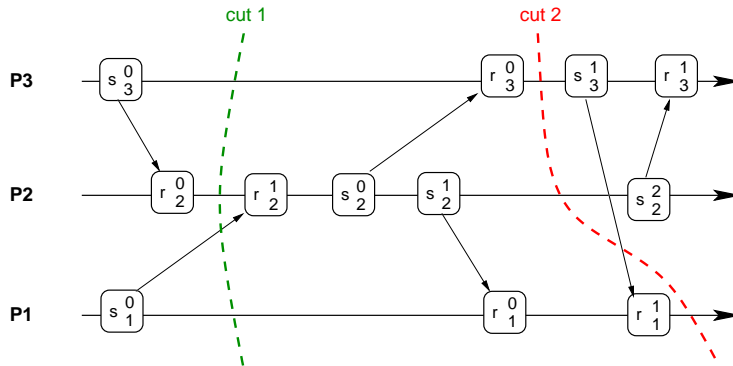


Figure 5: A consistent and inconsistent cut

We call a cut *consistent* iff for all events $e' \in C$, $e \rightarrow e'$ implies $e \in C$ (i.e., all events that happened before are also in the cut). A global state is consistent if it corresponds to a consistent cut. As a result, we can characterise the execution of a system as a sequence of consistent global states $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \dots$. Figure 5 displays both a consistent cut (labeled “cut 1”) and an inconsistent cut (labeled “cut 2”). For the inconsistent cut note that the event that happened before r_1^1 (i.e., s_3^1) is not part of the cut.

A global history that is consistent with the happened-before relation \rightarrow is also called a *linearisation* or *consistent run*. A linearisation only passes through consistent global states. Finally, we call a state S' is *reachable* from state S if there is a linearisation that passes through S and then S' .

Snapshots

Now that we have a precise characterisation of a consistent cut, the next question is whether such cuts can be computed effectively. Chandy & Lamport [CL85] introduced an algorithm that yields a *snapshot* of a distributed system, which embodies consistent global state and takes care of messages that are in transit when the snapshot is being performed. The resulting snapshots are useful for evaluating stable global properties.

Chandy & Lamport’s algorithm makes strong assumptions about the underlying infrastructure. In particular, communication must be reliable and processes be failure-free. Furthermore, point-to-point message delivery must be ordered and the process/channel graph must be strongly connected (i.e, each node can communicate with every other node). Under these assumptions, and after the algorithm completes, each process hold a copy of its local state and a set of messages that were in transit, with that process as their destination, during the snapshot.

The algorithm proceeds as follows: One process initiates the algorithm by recording its local state and sending a *marker message* over each outgoing channel. On receipt of a marker message over incoming channel c , a process distinguishes two cases:

1. If its local state is not yet saved, it behaves like the initiating process and saves the local state and sends marker messages over each outgoing channel.
2. Otherwise, if its local state is already saved, it saves all messages that it received via c since it saved its local state and until the marker arrived.

A process' local contribution is complete after it has received markers on all incoming channels. At this time, it has accumulated (a) a local state snapshot and (b), for each incoming channel, a set of messages received after performing the local snapshot and before the marker came down that channel.

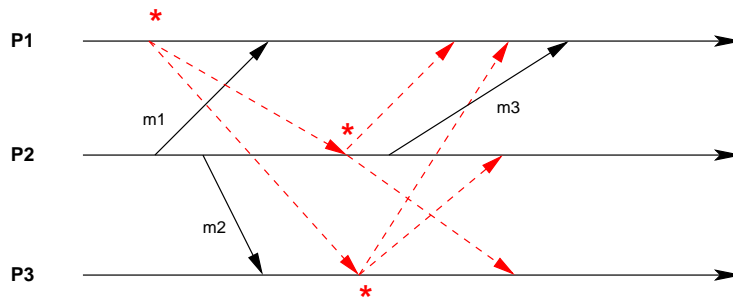


Figure 6: Marker messages during the collection of a snapshot

Figure 6 outlines the the marker messages (dotted arrows) and points where local snapshots are taken (marked by the stars) for three processes.

Distributed Concurrency Control

Some of the issues encountered when looking at concurrency in distributed systems are familiar from the study of operating systems and multithreaded applications. In particular dealing with race conditions that occur when concurrent processes access shared resources. In nondistributed system these problems are solved by implementing *mutual exclusion* using local primitives such as locks, semaphores, and monitors. In distributed systems, dealing with concurrency becomes more complicated due to the lack of directly shared resources (such as memory, CPU registers, etc.), the lack of a global clock, the lack of a single global program state, and the presence of communication delays.

Distributed Mutual Exclusion

When concurrent access to distributed resources is required, we need to have mechanisms to prevent race conditions while processes are within *critical sections*. These mechanisms must fulfill the following three requirements:

1. *Safety*: At most one process may execute the critical section at a time
2. *Liveness*: Requests to enter and exit the critical section eventually succeed
3. *Ordering*: Requests are processed in happened-before ordering

Method 1: Central Server

The simplest approach is to use a central server that controls the entering and exiting of critical sections. Processes must send requests to enter and exit a critical section to a lock server (or coordinator), which grants permission to enter by sending a *token* to the requesting process. Upon leaving the critical section, the token is returned to the server. Processes that wish to enter a critical section while another process is holding the token are put in a queue. When the token is returned the process at the head of the queue is given the token and allowed to enter the critical section.

This scheme is easy to implement, but it does not scale well due to the central authority. Moreover, it is vulnerable to failure of the central server.

Method 2: Token Ring

More sophisticated is a setup that organises all processes in a *logical* ring structure, along which a token message is continuously forwarded. Before entering the critical section, a process has to wait until the token comes by and then retain the token until it exits the critical section.

A disadvantage of this approach is that the ring imposes an average delay of $N/2$ hops, which again limits scalability. Moreover, the token messages consume bandwidth and failing nodes or channels can break the ring. Another problem is that failures may cause the token to be lost. In addition, if new processes join the network or wish to leave, further management logic is needed.

Method 3: Using Multicast and Logical Clocks

Ricart & Agrawala [RA81] proposed an algorithm for distributed mutual exclusion that makes use of logical clocks. Each participating process p_i maintains a Lamport clock and all processes must be able to communicate pairwise. At any moment, each process is in one of three states:

1. **Released:** Outside of critical section
2. **Wanted:** Waiting to enter critical section
3. **Held:** Inside critical section

If a process wants to enter a critical section, it multicasts a message $\langle L_i, p_i \rangle$ and waits until it has received a reply from every other process. The processes operate as follows:

- If a process is in **Released** state, it immediately replies to any request to enter the critical section.
- If a process is in **Held** state, it delays replying until it is finished with the critical section.
- If a process is in **Wanted** state, it replies to a request immediately only if the requesting timestamp is *smaller* than the one in its own request.

The only hurdle to scalability is the use of multicasts (i.e., *all* processes have to be contacted in order to enter a critical section). More scalable variants of this algorithm require each individual process to only contact subsets of its peers when wanting to enter a critical section. Unfortunately, failure of any peer process can deny all other processes entry to the critical section.

Comparison of Algorithms

When comparing the three distributed mutual exclusion algorithms we focus on the number of messages exchanged per entry/exit of the critical section, the delay that a process experiences before being allowed to enter a critical section, and the reliability of the algorithms (that is, what kinds of problems the algorithms face).

The centralised algorithm requires a total of 3 messages to be exchanged every time a critical section is executed (two to enter and one to leave). After a process has requested permission to enter a critical section it has to wait for a minimum of two messages to be exchanged (one for the current holder to return the token to the coordinator and one for the coordinator to send the token to the waiting process). The biggest problem this algorithm faces is that if the coordinator crashes (or becomes otherwise unavailable) the whole algorithm fails.

For the ring algorithm, the number of messages exchanged per entry and exit of a critical section depends on how often processes need to enter the section. The less often processes want to enter, the longer the token will travel around the ring, and the higher the ‘cost’ (in terms of messages exchanged) of entry into a critical section will be. With regards to delay, depending on where the token is it will take between 0 and $n - 1$ messages before a process can enter the critical section. The biggest problems faced by this algorithm are loss of the token and a crashed process breaking the ring. It is possible to overcome the latter by providing all processes information about the ring structure so that broken nodes can be skipped.

Finally, the decentralized algorithm effectively requires $2(n - 1)$ messages to be sent per entry and exit of a critical section (i.e., $n - 1$ request messages and $n - 1$ replies). Likewise there is a delay of $2(n - 1)$ messages before another process can enter the critical section (once again because $n - 1$ requests and $n - 1$ replies have to be sent). With regards to reliability the decentralised algorithm is worse than the others because the failure of *any* single node is enough to break the algorithm.

References

- [CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3:63–75, 1985.
- [Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, 1978.
- [RA81] G. Ricart and A. Argawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1), January 1981.