Master Thesis:
Animations and Analysis of Distributed Algorithms

Boris Koldehofe

12th April 1999

Contents

Chapter 1

Introduction

Distributed algorithms are by nature difficult to understand due to the existence of asynchronous threads of control that interacts and due to the lack of global state and time. Even if an algorithm starts from the same initial system configuration, it may not result in the same output. The animation of an algorithm shows graphically its execution, thus it can assist in understanding the behaviour of algorithms concerning

1. ”key ideas” of functionality of the algorithm
2. their behaviour under different timing and traffic of the system
3. their communication and time complexities

This thesis focuses on four classes of distributed algorithms, namely: *broadcast algorithms*, *spanning tree algorithms*, *resource allocation* and *synchronization algorithms* and *counting network algorithms*. Besides studying and investigating these classes, work done in the context of the thesis includes implementations of the algorithms as well as design and implementation of animation programs for them. It will be explained how animations for these algorithms were built and how they are helpful regarding functionality, behaviour and complexity of each algorithm.

The introductory part explains shortly the main concepts which are used for this thesis. In chapter 2 a general structure for building animations is given. The subsequent chapters 3, 4, 5 and 6 cover the previously mentioned areas of distributed algorithms. Every algorithm is explained and analyses. They show how the respective animation was designed and how the analysis results are represented inside the animation by also considering the general structure applied to animations according to chapter 2. Finally, chapter 7 gives basic information about the implementation itself.

Animation and implementation of the algorithms are part of an environment called LYDIAN [12] in which an algorithm can be simulated and its execution can be animated. The user has the possibility to change the structure of the network and the timing of links between processes so that any possible execution of the algorithm can be seen. For the purpose of changing network structures a modified *graphwin* of the *library of enhanced data structures and algorithms (LEDA)* [10] is used. It offers an easy way of constructing graphs and helps the user specifying timing assumptions for the network of processes which is represented by the constructed graph.

In order to built the animations itself, a library called POLKA [14] is used. This library supports many important features for building visualizations like multiple views, animation speed tuning, step-by-step execution and call-back events to assist interactive animations.

1.1 Distributed System and Models

A *distributed system* can be defined as a collection of autonomous processors, processes or computers which can communicate with each other through a communication medium (see, e.g. G. Tel [15]). The communication between processes can be done either through a shared memory medium ($\rightarrow$ *shared memory model*) or through *links* which interconnect processes directly with each other ($\rightarrow$ *message passing system*).

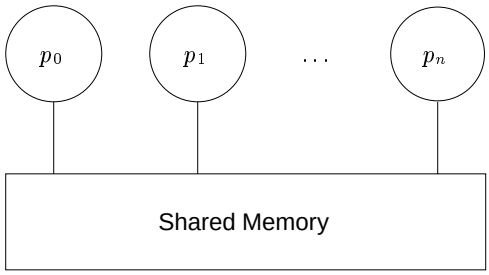

Figure 1.1: a shared memory model

In the shared memory model (cf. figure 1.1) processes can access in parallel memory locations which they share with all other processes. The communication between processes is achieved by writing information to common memory locations. In the message passing model (cf. figure 1.2) each process can read and write information only to a local memory; thus it must exchange information by sending messages via links to other processes.

The described and animated algorithms in the following chapters are all based on a message passing implementation. It is assumed that all processes of the distributed

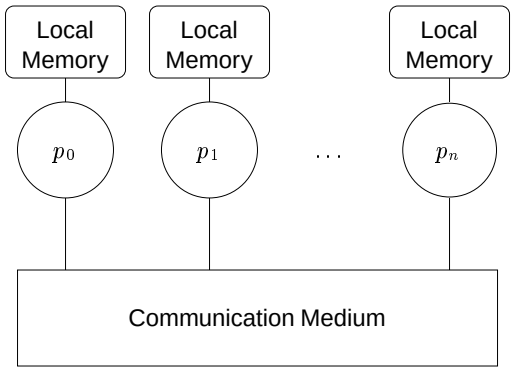

Figure 1.2: a message passing model

system execute the same algorithm and work correctly for any possible interconnection of processes. In this case the distributed system is described by a graph $G(V, E)$, called *communication graph*, where nodes in V represent processes and edges $(u, v) \in E$ represent a link from u to v. Through a link (u, v) process u can send messages to process v. Each process knows its state and can register following events:

- send event: a message is produced i.e it is sent to another process
- receive event: a message that was sent on a link is consumed by a process
- internal event: local computation

On an event a process performs a transition which means that it changes its state and might trigger some new events.

The described system is said to be an *asynchronous message passing system* if messages can be sent and received on links at arbitrary time. If a sender is only allowed to send a message on a link when the receiver is ready to receive this message then the described system is said to be a *synchronous message passing system*.

1.2 Complexity Measures and Causality

The quality of correct algorithms is analyzed by their complexity measures. In order to measure the complexity of a distributed algorithm in a message passing system one usually considers the *time complexity* and the *communication complexity*.

The time complexity measures the maximum time the algorithm needs in the worst case starting with its initialization until it comes to a halt. For this purpose an idealized timing is used in which the time is measured in message transmission units. Hereby it is assumed that the transmission of a message takes at most one time unit and the time which a process task needs to proceed is encapsulated in the time unit. Hence, the longest chain of dependent events will give a measure for time the algorithm was working as the execution is event driven. Often the analysis includes the upper bounds for a process task to proceed and a message to traverse on a link. However, it should be remarked that in real systems it is not possible to guarantee an upper bound for messages traversing on a link.

The communication complexity measures the traffic load of the system. This is achieved by counting the total number of messages that are exchanged among the system in the worst case. Also the size of messages might be interesting if it is not constant.

Causal relations are interesting in distributed computing due to the lack of global state in an asynchronous system. They show for each process which events were caused by an event of another process e.g. in a message passing system: the event of receiving a message will cause the process to do some local computation and/or sending messages to some other processes.

By computing local clocks for each process (see Lamport [8]) it is possible for a monitoring process to compute an order of all events that happened in the system, respecting the causal relations. This order needs not necessarily to be the same order in which the events really occurred, but it will lead to the same result of computation.

1.3 The LYDIAN environment

The simulation of distributed algorithms provides the possibility to test the behaviour of the algorithm under different timing and interconnection of processes. Its animation shows the algorithm's execution and allows to retrace this execution as well. Further it assists in the analysis by showing the algorithms complexity and causality.

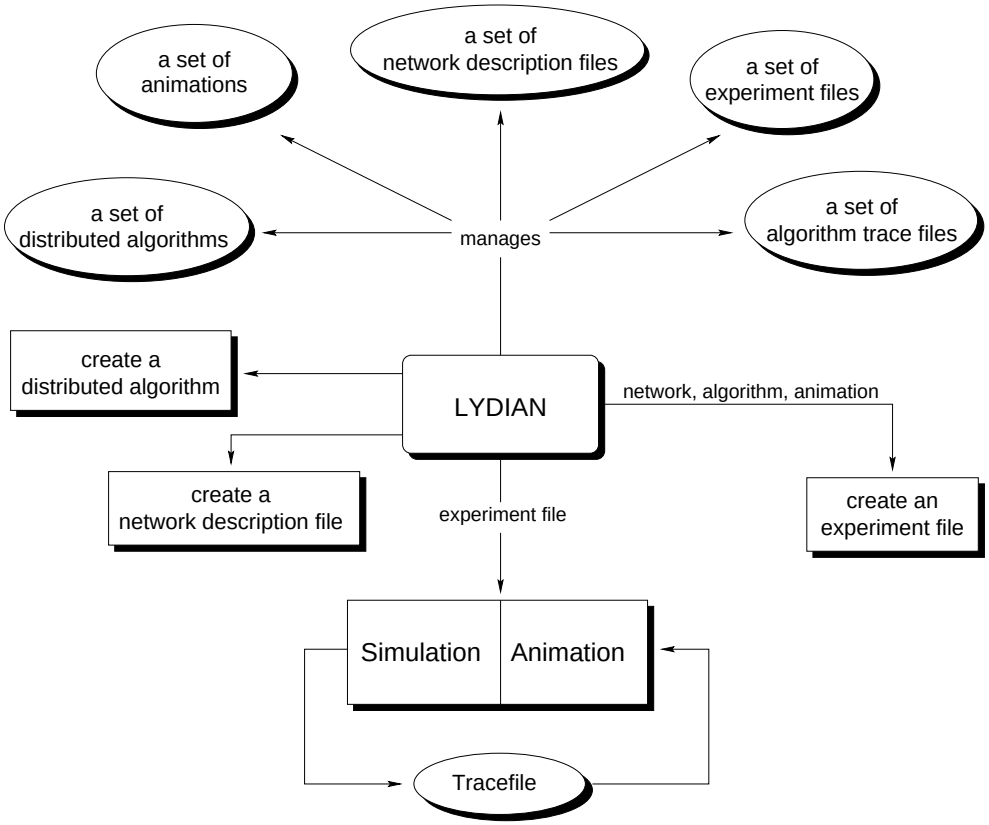

Figure 1.3: Overview of LYDIAN

LYDIAN [12] is an user friendly environment for the simulation and animation of distributed protocols (see overview in figure 1.3). Users can create their own experience files which consist out of a distributed algorithm, a network description file, an animation and a debug file for collecting information about the algorithm's execution. LYDIAN offers a set of implemented algorithms which can be selected by the user, but new algorithms can also be added. Further the user can choose among existing network description files or create some new. The network description file contains the whole communication graph including information about initialization, message transmission times among links and time which a process need to proceed from one state to another. Also the user must specify which kind of time model is associated with the algorithm (e.g. synchronous or asynchronous). As a small part of this thesis a new interface is implemented by using graphwin from the LEDA library [10] for the creation of network description files.

With the experiment file it is possible to start a simulation and view the results inside the debug file. In order to help the user in understanding these results an animation assists in reproducing the main events of the simulated algorithm. The construction of such animations was a main part of this thesis.

Chapter 2

Structure of the Animations

The introduced animation programs were built such that a program expects an input of important algorithm events and the time when an event happened. These events can be received either from a trace file or online during the evaluation of the algorithm. According to the receiving of those events the animation of the algorithm proceeds. The user is able to control the speed of the animation via a window called *Control Panel*.

In a distributed algorithm different aspects (e.g. main idea of the algorithm, communication complexity etc.) of the algorithm that needs to be shown will lead to different animation ideas. Therefore, this work suggests to use more than one animation window, called *view*, such that each view shows one of these aspects. On the other hand it is required to avoid confusion caused by showing too much at the same time, thus the user can select views, considered to be important, inside a window called *Animation Control Window*. The following set of views is available:

- The *Basic View* shows the main idea of the algorithm.
- The *Communication View* shows per process the traffic (messages) induced by the algorithms execution.
- The *Causality View* shows causal relations between events in the system execution.
- The *Process Step View* displays for a selected process its status information
- The *Process Occupation View* shows in actual time the time-period for each process when it was kept busy.

The animation of all views evolves simultaneously. At any time the user can change the selection of shown views. Some animations might not fit in the predefined window frame. Therefore each view offers possibilities to change the window size, move to different regions of the animation and to zoom in or out the displayed part of the animation. The next sections will give a deeper description of each view.

2.1 Basic View

The *basic view* illuminates the main idea of the algorithm by showing the user its most significant features. For a message passing system these are given by the states of processes and the exchange of messages. Depending on each algorithm further relation between processes will be shown e.g. for a resource allocation algorithm (described in chapter 5) it is important to see which process is owning which kind of resources.

An animation of the basic idea often shows the communication graph of the network. Nodes are represented by circles and edges by polylines between two nodes. Different colours are used in order to distinguish between states of processes and links. Sending a message is visualized by an arrow which points from sender to receiver and is continuously changing its size along the edge connecting sender and receiver processes. The arrow appears as long as the accordant message is not received. Since an algorithm may use different types of messages, for each message type a different colour is selected. Sending more than one messages is indicated by the arrow flashing between the colours of messages that have been sent along a link.

By their nature, algorithms can differ a lot, so this view is designed for algorithms in a different manner.

2.2 Communication View

The *communication view* shows the contribution of each process in the traffic induced by the algorithm's execution. The user can see a bar chart where each bar indicates the number of messages which have been sent for a process. An additional bar shows the average number of messages per process. The chart bar is useful regarding the message complexity of the algorithm. Often the user will easily find for processes a connection with their induced traffic. The bars grow online so that the user can see when traffic is induced.

In some algorithms the size of messages is not constant. Therefore for every process a circle indicates the size of the longest message that has been sent. The area of the circle is proportional to the size of the message. In order to recognize better the message sizes their number of bits are written below each circle. The variation of message sizes is also animated online, thus one is able to observe how fast the message size is increasing. In several algorithms, e.g. the ones that employ time stamping, this is important (see also chapter 5.2).

This view is implemented in a way that it can be used for all animations. The implementation allows flexibility in the way of counting messages. For some algorithms it might be better counting only some significant messages. Also a different colouring of bars and message sizes is allowed.

2.3 Causality View

In the *causality view* causal relations between processes are shown by an arrow pointing from process p, which caused event e, to process q, which was driven by e. The

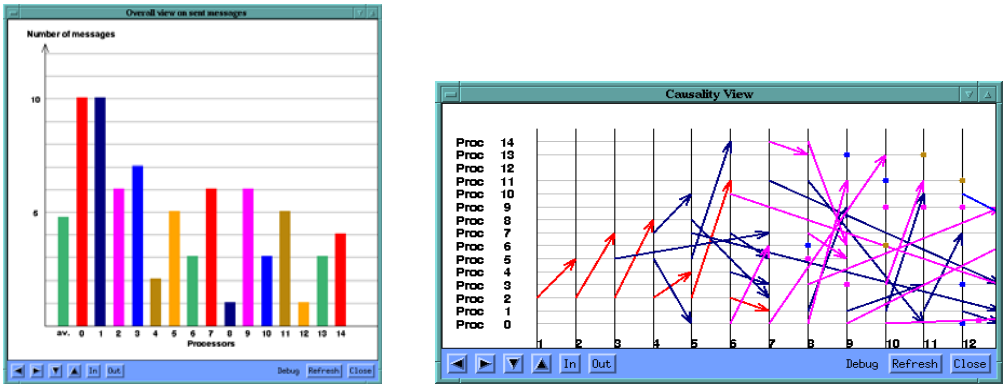

Figure 2.1: Communication view and causality view

arrow starts at the local clock's time of p, when e was originated, and stops at the local clock's time of q, when e was received. The local clock's time of processes is computed according to the computation of local clocks by Lamport [8]. Receiving a message causes a process to set its local clock value to

$$\max(\textit{send\_tm}, \textit{old value of local clock}) + 1$$

where *send_tm* denotes the local clock value of the sender at the time it sent the message.

The view shows how a monitoring process would see the current execution of the algorithm. The longest directed path gives an upper bound for the length of the execution in units of message transmission times. The implementation allows to use this view in all algorithms for which the colouring of causal relations can be shown in different ways.

2.4 Process Step View

The *process step view* helps the user to understand an execution of an algorithm by giving him the latest status of a selected process. The status contains information about identifier, state, latest event, latest message sent or received, local clock and real time. It is possible to select the process interactively by clicking with the mouse on the respective node inside the basic view. Furthermore the user can retrace the whole execution for a process by selecting interactively inside the process step view previous or next status information. Although algorithms use different states and events, the outfit of the view can be used by all animations.

2.5 Process Occupation View

The *process occupation view* shows in real time i.e. time given by the simulation trace the period when each process was kept busy by the algorithm. The user can conclude how the computation is distributed among the system. The occupation time

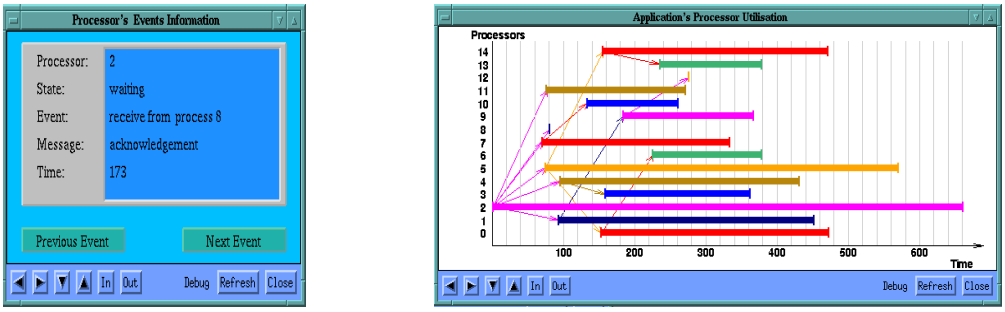

Figure 2.2: Process step view and process occupation view

is shown for a process by a bar that increases online as the algorithm evolves. The implementation allows all animations to use different colours for bars. Usually this is helpful to distinguish between processes when the algorithm needed such a long time that the whole animation does not fit in a window frame and the user must move to different regions. In some cases it is also significant to see how long a process was kept busy in a certain state. Therefore a bar can be split in sections which are distinguished by different colours chosen according to each algorithm. Further some important causal relations can be displayed by an arrow which can also be coloured differently.

Chapter 3

Broadcast

A *broadcast algorithm* is a basic algorithm used inside many other distributed algorithms e.g during initialization of a network where all processes should be waken up. It distributes an information known by a single process to all other processes of the network. Often the process initiating the broadcast algorithm needs to be acknowledged that all processes received the broadcasted information. Algorithms satisfying this property are called *broadcast with acknowledgement* algorithms.

In the following sections two broadcast algorithms for a message passing system are introduced. The underlying communication graph representing the network is required to be connected, since it is impossible for two unconnected components of the network to communicate. For the analysis of the algorithms l denotes the upper bound given for the time that any process task needs to proceed and d denotes the upper bound given for the time that any message needs to traverse on a link i.e the time between sending and receiving of a message. Further, n denotes the size, D denotes the diameter and δ denotes the degree of the communication graph.

3.1 A Broadcast Algorithm

This algorithm is an straight forward way of performing a broadcast. The initiator sends to all its neighbours a message of kind *broadcast*. When a process receives message *broadcast* for the first time, it sends to all other adjacent processes further *broadcast* messages.

Time Complexity

After $l + d$ time units the distance between the initiator and the processes which have not received the broadcasted information increases at least by one. Thus it takes at most $O(D(l+d))$ time units until every process received the broadcasted information.

Message Complexity

Every process needs to send at most one message on a edge which results in a total of $O(|E|)$ sent messages.

3.2 A Broadcast with Acknowledgement Algorithm

The described *broadcast with acknowledgement* algorithm of this section is an extension of the previous algorithm. As before the initiator sends messages of kind *broadcast* to all its neighbours. Also processes that receive message *broadcast* for the first time still send to all other adjacent processes further *broadcast* messages. In addition to this a process marks the sender of the first received *broadcast* message as its parent. Every process keeps track on *broadcast* messages it sent to adjacent processes by storing the receiver of each message in a set of expected acknowledgements, called ack. As long as ack is not empty the process waits to receive acknowledgements from processes stored in this set. On receiving an acknowledgement a process deletes the sender from ack. When ack is an empty set then all acknowledgements are received. Then a process sends an acknowledgement to its parent node and terminates the algorithm. Processes which receive a message *broadcast* although they have already decided for their parent node reply with an acknowledgement. This guarantees that every *broadcast* message will be answered by an acknowledgement. The algorithm is finished when the initiator received all acknowledgements.

Time Complexity

According to previous algorithm every process received the broadcasted information in time $O(D(l+d)$. Due to asynchrony the path on which an information is broadcasted might be of length $O(n)$ as messages might proceed on some links much quicker than messages on the shortest path (see example in figure 3.1).

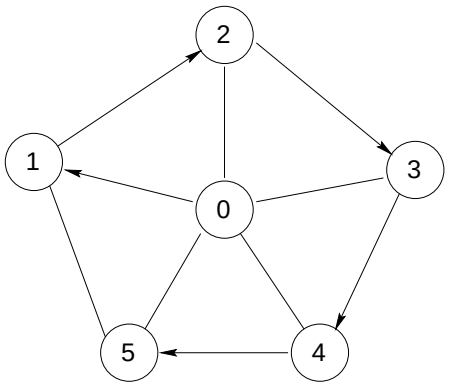

Figure 3.1: Example for a graph with diameter 2. Process 5 is guaranteed to receive a broadcasted message from process 0 in time $d+l$, but it is still possible that process 5 received the first *broadcast* message along the path indicated by the arraows.

In the convergecast phase of the algorithm the distance between the initiator and processes having received all acknowledgements decreases by at least one after time $d+l$, so it takes $O(n(d+l))$ until the initiator has received all acknowledgements. Hence, the whole algorithm is guaranteed to terminate in time $O(n(d+l))$.

Message Complexity

Every process sends along each adjacent link at most one *broadcast* and one *acknowledgement* messages. For this reason a total of $O(|E|)$ messages are sent along the network during the execution of the algorithm.

Conclusion

It should be noted that this algorithm also computes a spanning tree which is formed by the edges on which a process receives its first *broadcast* message. This spanning tree can be useful for further broadcasts. Then each process receives at most one *broadcast* message and sends one acknowledgement. Hence, the message complexity is reduced to $O(n)$.

3.3 Animation of the Broadcast Algorithms

The broadcast with acknowledgement algorithm is an extension of the broadcast algorithm introduced in section 3.1. Therefore this section introduces only an animation for the broadcast with acknowledgement algorithm. This animation was built according to the proposed structure in chapter 2. In the following the usage of the introduced views is described:

Basic View

The basic view (cf. figure 3.2) shows the communication graph of the network. By default all processes are shown as yellow circles. As the animation proceeds processes will change their colour according to their state:

- A process in state sleeping i.e. it has not started running the algorithm is coloured yellow.
- The initiator of the algorithm is coloured red.
- A process which received some *broadcast* messages, but still waits for acknowledgements is coloured green.
- A process which received all acknowledgements is coloured blue.

With each process a unique identifier is associated in order to help the user to recognize this node in other views.

Links usually appear as black polylines, but will change their appearance when a link is determined as a spanning tree edge. These are links on which a process received its first *broadcast* message. The link will change its shape into a red arrow which points from sender to receiver of the accordant *broadcast* message. At the end of the animation the user can see a complete spanning tree and observe the longest path from initiator to any other process. This determines the worst case execution time.

Messages are shown by arrows which point from sender to receiver. As long as a message is transmitted the respective arrow is continuously changing its size along the

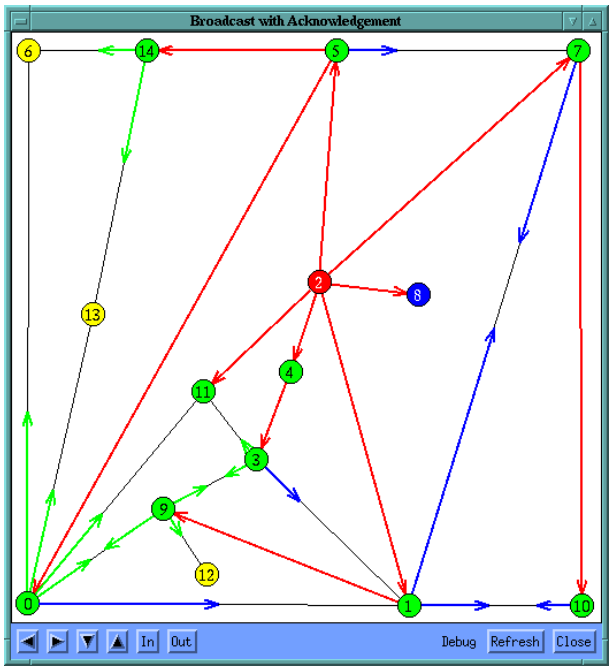

Figure 3.2: Basic view of the broadcast-with-acknowledgement algorithm

link connecting sender and receiver of the message. Messages *broadcast* are coloured green while messages *acknowledgement* are coloured blue. If two messages, one of kind *broadcast* and the other of kind *acknowledgement*, are sent along the link at the same time the arrow, representing these messages, will start flashing between green and blue.

Communication View

The communication view counts for each process the number of sent messages. Moreover it shows the average number of messages which are sent by a process. Although the message size is constant, the message size is displayed below every process as well.

For this algorithm it is easy to observe that the number of sent messages is proportional to the degree of a node. Hence, an initialization of this view assuming that for each process the message size is bounded by 2δ messages will guarantee an optimum scale for this view.

Causality View

The causality view shows the causal relations of the algorithm. A relation is shown in form of an arrow pointing from sender to receiver of a message beginning at *send_tm* and ending at *rec_tm*. Hereby *send_tm* denotes the local clock of the sender when it sent the message, while *rec_tm* denotes the local clock of the receiver when receiving the message. The local clock of a process is updated when a message was sent or

received. Before sending a message a process increases its local clock by one, while receiving a message causes a process to set its local clock to

$$\max(\textit{send\_tm}, \textit{old value of local clock}) + 1.$$

The causal relations are coloured according to a *local colour value*. Initially this value is 0 for all processes. The local colour value of a process is updated when a message is received. Then the process updates its local colour value to

$$\max(\textit{own local colour value}, \textit{sender's local colour value}) + 1.$$

Process Step View

The process step view has the same appearance as in all other animations. A user can click on a process inside the basic view in order to see inside the process step view information about latest state, event, time and local clock. For every process the user can retrace the sequence of events by clicking inside this view on buttons "Previous Event" or "Next Event" (see also page 8).

Process Occupation View

This view shows in real time the period between processes start and stop participating at the algorithm i.e. a process starts participating when it sends *broadcast* messages and it stops participating when it received all acknowledgements. In order to illustrate which process invokes other processes, arrows pointing from sender to receiver are shown. They appear in the colour associated with the sender. The bars showing the process occupation are displayed according to the colour associated with the processes identifier.

Chapter 4

Minimum Spanning Tree

For distributed systems a *minimum weight spanning tree* is useful to reduce the costs for broadcasting information along the network. In the previous chapter it is observed how a spanning tree helps reducing the message complexity of a broadcast algorithm. A minimum weight spanning tree minimizes the total cost of edges selected for the spanning tree. Hence, it minimizes the cost of a broadcast that uses tree edges, if the weight of each edge represents the cost of sending the information over the respective link.

Let $G(V, E)$ be an undirected graph where V denotes a set of nodes and E a set of edges. A subgraph $G'(V, E')$ of G which contains no cycles, is defined to be a *spanning forest*. A spanning forest $G'(V, E')$ of G which is also connected is said to be a *spanning tree* of G. For every edge $e \in E$ let be a weight w_e associated. The spanning tree $G'(V, E')$ of G is called a *minimum weight spanning tree (MST)* when for every other spanning tree $G''(V, E'')$ of G

$$\sum_{e \in E'} w_e \leq \sum_{e \in E''} w_e$$

is valid.

An easy way of computing a MST is to sort the edges of E in ascending order according to its weights. Initially, each node is one set called component. The algorithm by Kruskal [7] takes each edge $(u, v) \in E$ in the order they were sorted and checks whether nodes u and v belong to the same component. If u and v belong to different components (u, v) will be an edge of the MST and the components of u, v and edge (u, v) will be united. If an union find implementation with path compression is chosen the MST is computed in time $O(|E| \log |E|)$ (cf. analysis in the book of Cormen, Leiserson and Rivest [4]).

The algorithm uses the invariant that every component created by the algorithm is part of a MST. For two components united on edge e, it is assured that e is the minimum outgoing edge of both components since e is of minimum weight among all by the algorithm not touched edges. Further lemma 4.1 will show that every component united with its minimum outgoing edge will be part of a MST. Thus, the unification between both components must also be part of a MST.

Lemma 4.1 *Let $G'(V', E')$ be an arbitrary acyclic and connected subgraph of G(V,E) where $V' \subseteq V$. If the edges of E' are part of a MST then there exists a MST of G which*

contains all edges of E' and the minimum edge (u, v) with the property that $u \in V - V'$ and $v \in V'$.

Proof. Let T be the minimum spanning tree that contains E' and (u, v). Assume there exists a spanning tree T' with smaller weight than T which is also containing all edges of E', but not (u, v). If one constructs T from T' by adding (u, v) to its edges there will be at least one edge which has only one endpoint in V' causing a cycle. Otherwise T' would not be connected. These edges have weight greater or equal the weight of (u, v) because (u, v) was the minimum weighted edge with only one endpoint in V'. By removing these edges T becomes a spanning tree with weight less or equal the weight of T' which is a contradiction to the assumption that weight of T' is smaller than weight of T. □

4.1 The GHS Spanning Tree Algorithm

The idea of merging components is also used for the distributed implementation of the algorithm by R.A Gallager, P.A. Humblet and P.M.Spira [5]. There exists a synchronous and an asynchronous implementation of this algorithm which are both described according to N.Lynch [11]. Hereby the understanding of the synchronous part will give an easier understanding of the asynchronous part.

Modeling the MST problem within a communication graph

Let $G(V, E)$ be the graph for which the computation of a MST is required. It is assumed that there exists exactly one process for each node of V and exactly one link for every edge $(u, v) \in E$ between processes that represent nodes u and v. Every process knows about its own identifier which is different from all other processes and the weight of its incident edges. For simplicity reasons it is assumed that all edges have different weights although allowing edges to have the same weight can be resolved by applying a total order to the edges regarding identifiers of processes.

Basic Idea

The basic idea is inspired by the previous algorithm. In the beginning every process represents its own component, but in this algorithm all components try to expand at the same time without knowledge of an order in which edges are sorted by weight. Every component tries to find its *minimum outgoing edge (MWOE)* and to combine itself with the component adjacent to the MWOE. The algorithm terminates when there is only one component left.

The parallel unification brings new problems. According to lemma 4.1 the components which are united along the MWOE of one component will be part of a MST. Due to the parallel unification of components it is possible that cycles among the united components occur e.g. a complete graph of three nodes A, B, C where each edge is of weight one. Thus component A could be combined with component B, while B

is combined with C and C is combined with A. However, cycles can be avoided by assuming that all edges are of distinct weights.

Lemma 4.2 *If all edges $\in E$ of a graph $G(V, E)$ have distinct weights, then there will exist exactly one MST for G.*

Proof. Assume there exists two MSTs T and T' of G. Let e denote the minimum weighted edge which is in T, but not in T'. Then $T'' = T' \cup e$ will form a cycle in T'' with at least one edge e' which is not in T. Due to the choice of e, e' must have a greater value than e. Removing e' will result in a tree of smaller weight than T', a contradiction that T' was a minimum spanning tree. □

4.1.1 Description of the Synchronous GHS Spanning Tree

The algorithm constructs the MST in levels. Initially in level 0 every component is a single process. Inductively each component of level k is built out of components of level $k-1$ such that it is guaranteed to have at least 2^{k-1} processes. The levels are built synchronized which means that after round k only level k components exist. Each round is guaranteed to be finished in time $O(n)$, where n is the number of processes.

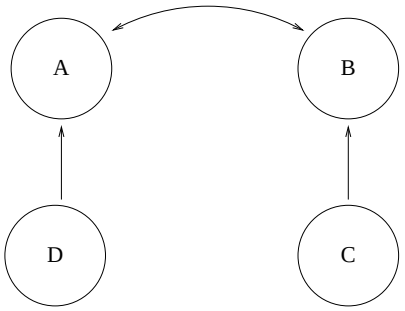

Figure 4.1: Example of a combination of four level k components which are going to form a new level $k+1$ component. The edges represent MWOE pointing from its component to the component they are connecting.

It is assumed that every process knows a *unique identifier (UID)* and which of its adjacent edges belong to the component's tree. Further every component has a leader and all processes of the component knows the leader's UID. In round k all level k components try to find their MWOEs. This is realized by the leader broadcasting to all processes of the component the information to search for their local MWOE along their non tree edges i.e. nodes of the components spread *initiate* messages along tree edges and *test* messages along all non-tree-edges. The synchronization guarantees that *test* messages will be sent at the time when all members of the component received *initiate* messages. The *initiate* message includes also information about the leader's UID so that in every level all processes are informed which process is the component's leader. Similar to the way acknowledgements are convergecast to the initiator of a broadcast with acknowledgment algorithm (see chapter 3), the local MWOE is convergecast to

the leader which determines the new MWOE. The leader sends message *change-root* to the component's process along the MWOE. This message induces the receiver to determine a new leader by sending message *connect* along the MWOE. Both components will be combined along the MWOE by marking this edge as a tree edge. It should be noted that more then two components can be combined in one step as one can see in figure 4.1, but only for one pair of components inside a set of combined components the MWOE is the same edge. Along such an edge the node with lower UID is chosen as the leader of the new component. An edge with such a property can be determined locally by the adjacent processes since two *connect* messages must be sent along this edge.

Hence, it is possible two describe the combining process by two possible scenarios. In the first scenario the MWOE of component A is also the MWOE of component B. In this case A and B will be united and the node with lower UID along the MWOE will be the new leader of the new component. The new UID of the leader will be broadcast together with the *initiate* message of the next round.

In the second scenario, let e be connecting components A and B and e be a MWOE of A, but not of B. In this case A will be absorbed by B and e will be known as an tree edge connecting components A and B. The *initiate* message of the next round will broadcast the UID of component Bs leader. Of course, component B will be united or absorbed along its MWOE as well.

Time Complexity

Due to both scenarios of combining components it is guaranteed that a level k component will consist out of at least 2^{k-1} processes. Hence it takes at most $\log n$ rounds until one single component remains and the algorithm terminates. Since a component cannot exceed the number of n processes, every broadcast within a round takes time at most $O(n)$ steps. After time $O(n)$ all nodes know to which component they belong and start sending *test* messages which are straight replied along their non-tree-edges. Convergecasting the local MWOEs to the leader and sending *change-root* message towards the MWOE will take another $O(n)$ steps, which implies a total bound of $O(n)$ steps per round. Therefore the whole algorithm is guaranteed to terminate in time $O(n \log n)$.

Communication Complexity

In every round each process sends messages to all its neighbours (*initiate* messages to tree-edges and *test* messages to non-tree-edges) in order to determine the MWOE. Thus $O(|E|)$ messages are necessary to compute MWOEs of all components. Another $O(n)$ messages are necessary to determine new leaders of components. Altogether this gives a bound of $O(\log n(n + |E|))$ for the total number of messages send during the execution.

This bound can be improved by marking non-tree-edges which are known to belong to the same component. Then *test* messages need only be sent to processes which are not known to belong to the same component. Further in each round edges are tested one after the other in increasing order according to their weights until an edge is found

which leads outside the component. Along tree-edges still a total amount of $O(n \log n)$ messages are sent. An amortized analysis is used in order to determine the number of messages along non-tree edges. Each tested edge gets rejected at most once, leading to a total of $O(|E|)$ messages. During each round at most one tested process is accepted by each process, which is a total of $O(n \log n)$ messages. Since each *test* message is either accepted or rejected altogether $O(|E| + n \log n)$ messages are sent along non-tree edges which implies a total bound of $O(|E| + n \log n)$.

Correctness

For correctness it must be shown that all determined MWOEs belong to the MST and that the algorithm guarantees progress. Further the leader of each component is required to be unique during each round.

For all edges distinct weights are applied. According to lemma 4.2 the MST of the graph is unique. Hence, MWOEs belong to the MST if their components are also part of a MST (see lemma 4.1). Initially all components are part of the MST and they are combined with other components on their MWOEs. Thus at each level a component is part of the MST. As long as there exists more than one component MWOEs can be determined. Thus the algorithm guarantees progress at each level.

Initially, all components consist out of a single process which implies a unique leader for level 0. In round k for each set of level k components, which are combined to a level $k+1$ component, a unique leader is selected along the MWOE, which is the MWOE of two components. The process with lower UID along this MWOE will be selected as the new leader. Lemma 4.3 will show that for such a set of level k components this MWOE is unique.

Lemma 4.3 *Let $G'(V', E')$ be a directed graph. Nodes of V' represent level k components which are part of the MST of graph $G(V, E)$ whose edges have distinct weights. Edges $(u, v) \in E'$ represent the MWOE of the component to which u belongs. For each subgraph of connected components there is exactly one unique cycle between two components.*

Proof. The MST is unique because of Gs distinct weights, and MSTs allow no cycles. Since all $MWOE$s are part of the MST, there can only occur cycles in G' of the form (u, v) and (v, u).

Assume there is no such cycle among a set of connected components. Then there exists one component which has no outgoing edges, a contradiction to every component has a MWOE.

Assume there are more such cycles among a set of connected components. Then there must exist a path between the pairs of cycling components. Hence one component must have more then one MWOE contradicting that every component has only a single MWOE. □

4.1.2 Description of the Asynchronous GHS Spanning Tree

The asynchronous GHS spanning tree algorithm is quite similar constructed to the synchronous version, but asynchrony leads to new problems which must be resolved. In the following these problems and solutions are described in order to give the reader an idea of its correctness, although a real proof of its correctness is omitted due to its length. One proof of correctness for this algorithm is shown in a paper by Welch, Lamport and Lynch [16].

Problems to be Resolved

The *first problem* arises for a process by checking along non-tree-edges whether adjacent processes belong to the same component. Synchrony guarantees that two processes communicating with each other can conclude from their leaders UID whether they belong to the same component. In the asynchronous case the process might not have received the UID of the latest leader when receiving a *test* message.

The *second problem* is caused by components which might grow with different speed because levels are not synchronized anymore. Therefore one could gain an increase in the communication complexity since combining a component constantly with level one components will lead to $\Omega(n^2)$ rounds and increases the number of messages which have to be sent. Also two neighbour components searching for their MWOE in different levels might cause unpredictable interference.

Solutions

The above described problems can be avoided when each node keeps track of its level. As in the synchronous version, the leader of a level k component broadcasts an *initiate* message including its UID, but also its level throughout its component. The nodes of the component update on receiving message *initiate* their entry for UID and level. It should be noted that in the following for a process p *UID*(p) denotes the UID of the leader known by p, while *level*(p) denotes the current known level information of p.

A process p that receives a *test* message by a neighbour process q together with q's level can conclude the following:

- If *UID*(p) = *UID*(q) then p and q belong to the same component as two nodes once belonged to one component will do so also in the future.
- If *UID*(p) $\neq$ *UID*(q) and *level*(p) $\geq$ *level*(q) then both nodes do not belong to the same component because a component proceeding to a new level implies that local MWOEs of previous levels are determined. Thus a node of higher level will never receive a *test* message from a node of the same component. If the levels of nodes are equal they must belong to different components since all processes of a level k-component know the same leader
- If *UID*(p) $\neq$ *UID*(q) and *level*(p) $<$ *level*(q) then p cannot decide whether it belongs to the same component as q. Therefore it delays its answer until *level*(p) = *level*(q).

These conclusions directly solve the first problem. The use of levels is also a solution to the second problem by merging only components which are of the same level and thus guaranteeing that a level k component has at least 2^{k-1} processes. However, a component of lower level can be absorbed by a component with higher or equal level, but a component cannot absorb a neighbour component of higher level since it might want to merge with this component when it reached a level of same height.

Progress

Components can reach different levels at a time so that it has to be reconsidered whether the algorithm is guaranteed to make any progress, an important feature for the correctness of the algorithm. Regarding components with lowest level will show that progress is still guaranteed. Since these components cannot be delayed by neighbouring components, they will manage to determine their MWOEs. If MWOEs lead to a higher level component an absorb operation will be performed. Otherwise, if for a connected set of these components the MWOE leads only to same-level-components then according to lemma 4.3 a merge operation is possible and a higher-level-component will be created. So after a finite number of steps all lowest-level-components will be part of higher level components. Thus latest at level $\log n$ there will be a single component and the algorithm terminates.

Time Complexity

Let l denote the upper bound given for the time that any process task needs to proceed and let d denote the upper bound given for the time that any message needs to traverse a link i.e time between sending and receiving the message. If all processes are woken up and each process determined a sorted order of its edges weights then lemma 4.4 will show by induction that the time for all processes to reach level at least k will be $O(kn(l+d)$. This implies time $O(n \log n(l+d))$ for the whole algorithm because a broadcast algorithm wakes up all nodes latest in time $O(n(l+d))$ and each process will determines a sorted order of its edges in time $O(nl \log n)$.

Lemma 4.4 *Assuming all process are woken up and each process determined a sorted order of its weight. The time for all processes to reach at least level k of the asynchronous GHS-MST algorithm is $O(kn(l+d))$.*

Proof. Initially all processes are level one components after are woken up. Thus all components reach in $O(1)$ steps the initial level. Assume that all processes reached at least level k in time $ckn(l+d) =: s$, with c denotes a constant. After the leader of a level k component started with broadcasting message *initiate* it takes time $n(l+d)$ until all processes of the component received this information. Therefore, after time $s + n(l+d)$ the last process starts sending *test* messages along their non-tree-edges. For each sent *test* message it takes at most time $2(l+d)$ until an answer is received. Note that there is no additional delay because all neighbour components are at least of the same level or even of higher level. A process will will send at most $n-1$ *test* messages one after the other such that the local MWOE is determined latest after time $s+3n(l+d)$ and after time $s+4n(l+d)$ all information are convergecast to the leader.

Sending messages *change-root* and one message *connect* takes at most additional time $n(l+d)$. So after time $5n(l+d)$ all MWOEs of all level k components must have been determined and due to the message *connect* these components are merged with each other or absorbed. Hence latest after time $c(k+1)n(l+d)$ all components will have progressed to level $k+1$ if $c \geq 5$ is chosen. □

Communication Complexity

The analysis, similar to the synchronous case, divides messages into two groups. Messages *test* which are answered by a *reject* message belong to the first group. A rejected *test* message, which was sent along (p,q) by process p, implies that p and q belong to the same component. Therefore p will send a *test* message which will be rejected at most once along an incident edge. Hence, $O(E)$ messages of the first group are sent during the algorithms execution.

The second group counts the accepted *test* messages, *initiate* messages, *report* messages and *change-root* messages. Every node of the component will receive in each level at most one accepted *test* message because it tests its adjacent edges one after the other in increasing order according to the weight of edges. Further each node of a component receives at most one *initiate* and one *change-root* message. Each *initiate* message will be replied by one *report* message. Then for a component C the number of messages, which are sent during its existence (which is until it is combined with other components), is $O(|C|)$ where $|C|$ denotes the number of processes that belong to C. Thus the total number of messages that belong to the second group is proportional to $\sum_C |C|$. This can be transformed to

$$\sum_{k=1}^{\log n} \sum_{C;level(C)=k} |C|$$

where *level*(C) denotes the components level. All level k components consist out of a distinct set of processes. Hence for each level k

$$\sum_{C;level(C)=k} |C| = n$$

which implies a total of $O(n \log n)$ messages of the second group. Therefore the algorithm needs to send $O(n \log n + |E|)$ messages.

4.1.3 The Detailed Asynchronous GHS Spanning Tree Algorithm

In the following a detailed description of the asynchronous GHS-spanning-tree is given. It includes a description of the different messages and the possible states of edges known by the adjacent processes. The algorithm is described in the way it was also implemented and later animated inside LYDIAN.

Messages:

- *Initiate:* An *initiate* message is broadcast throughout a component, starting from the leader, along the edges of the component's spanning tree. The message triggers processes to determine their local MWOEs.

- *Report:* A *report* message convergecast information about minimum-weight edges back towards the leader.

- *Test:* A process sends to its neighbour a *test* message when it wants to find out whether both belong to the same component. This is going to happen when nodes look for their local MWOE.

- *Accept/Reject:* These messages are used for responding on a *test* message. The process sends an *accept* message when the sender is in a different component and a *reject* message otherwise.

- *Change-Root:* After determination of the MWOE the leader of a component sends this message to the process which is incident to the MWOE and belongs to the same component. This process will attempt to combine with the component on the other side of the MWOE.

- *Connect:* This message is used when a process attempts to combine its component with the component of its adjacent process along the MWOE. If two *connect* messages were sent along the MWOE it means that a merge operation is performed such that the node along the MWOE with higher UID becomes the new leader. Otherwise the component that sent the *connect* message will be absorbed by the other.

State of edges

- *Branch:* An edge in state *branch* is associated to be an edge along the MST.

- *Rejected:* An edge in state *rejected* is determined not to be an edge of the MST since it connects a process with a neighbour of the same component.

- *Basic:* An edge in state *basic* is an edge which have not been classified in one of the above categories.

The Algorithm

Initially all processes are level 0 components.

1. Finding the MWOE:

Leader of a component: starting a new phase

send *initiate* messages containing information about components id (also called core) and level along its MST edges (edges in state *branch*).

as long as their exists edges in state *basic* (edges that have not been classified yet) and *test* messages along these edges are replied by a *reject* message, send a *test* message along the edge of minimum weight in state *basic* in order to find the local MWOE

wait for the convergecast of information of local MWOEs along MST edges

Processes: on receiving a message *initiate*

pass *initiate* message to all other MST edges

as long as their exists edges in state *basic* and *test* messages along these edges are replied by a *reject* message, send a *test* message along the edge of minimum weight in state *basic* in order to find the local MWOE

wait until all sent messages have been replied and compute out of this information the local MWOE

reply this information in form of a *report* message to parent process

Processes: on receiving a message *test*

if process and sender have the same component id
$\Rightarrow$ send *reject*

if process and sender don't have the same component id
and *level*$(process) \geq$ *level*$(sender)$
$\Rightarrow$ send *accept*

if process and sender don't have the same component id
and *level*$(process) <$ *level*$(sender)$
$\Rightarrow$ delay answer until *level(process)* $\geq$ *level(sender)* or both will have the same component id

2. Determine the new leader of the component

Leader: having been answered all *initiate* and *test* messages

if there is no MWOE $\Rightarrow$ the algorithm is finished

otherwise send a *change-root* message to the process that belongs to the component and is incident to the MWOE

Processes: on receiving a message *change-root*

if process is the target process of this message

send a *connect* message along the MWOE

if a *connect* message along the MWOE was received before (this is the case if this edge is already marked as a MST edge) and *UID(process)* $>$ *UID(sender)*
$\Rightarrow$ start next phase as the leader of the merged component

mark the MWOE as a MST edge

else pass message along the minimum weight edge (edge which leads to the MWOE)

Processes: on receiving a message *connect* on edge e

if the component of the sender has a lower level
$\Rightarrow$ absorb this component by sending message *initiate*

else if the state of e is *basic*
$\Rightarrow$ delay decision until it is clear whether e is also a MWOE of this component.

else if *UID(process)* $>$ *UID(sender)*
$\Rightarrow$ start next phase as the leader of the merged component

4.2 Animation of the GHS Spanning Tree Algorithm

The structure of this animation relies on the general structure introduced in chapter 2. All proposed views (Basic View, Communication View, Causality View, Process Step View and Process Occupation View) are implemented. A description of the animation ideas for each view is given in the following.

Basic View

The basic view (cf. figure 4.2) shows the communication graph of the network which is at the same time the graph for which a MST should be computed. Nodes and edges are coloured according to their states. Messages are shown by arrows which are pointing from sender to receiver and are resizing along this edge. Each message is represented by a different colour. If more than one messages are sent along a link the arrow representing these messages will start flashing between the colours of the sent messages. In the following the colouring and shapes of the animation items is listed:

- *Nodes:* All nodes are by default a yellow circle. Each node has a unique identifier which helps to recognize this node in other views. If a node becomes leader of a component then it changes its colour into red. A leader of a component which tries to combine along one of its adjacent edges with another component, changes its colour into orange. This way it shows that it is not decided whether this node will also be leader of the combined component. A node coloured in orange will be coloured yellow again when this node receives a message *initiate*. This means there must exist another leader of the component. Hence this node cannot represent the leader any longer. A red node v will be coloured yellow when an adjacent node received a message *change-root* sent by v.

- *Edges:* The default shape of all edges is a black polyline connecting two nodes. An edge will be changed into a thick red polyline when it was determined as a tree edge of the MST. During the combination process of two components, the edge connecting these components is shown as a thick red dotted polyline. The combination process is finished when an *initiate* message is sent along this edge such that this edge changes into a solid thick red polyline.

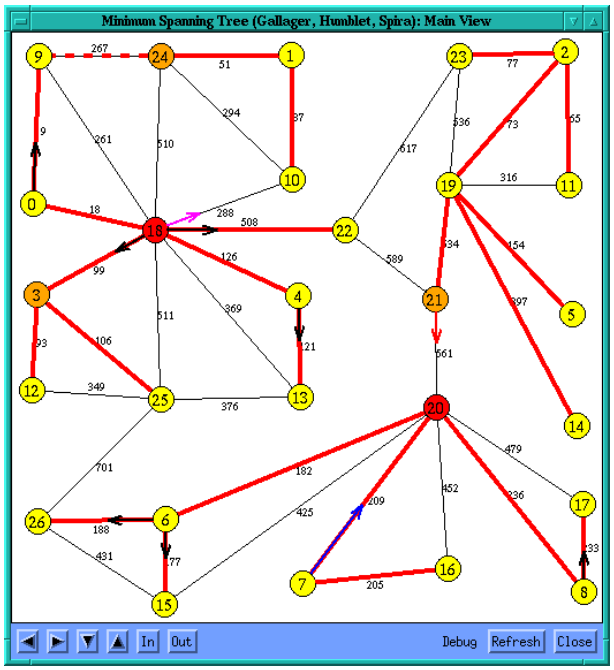

Figure 4.2: Basic view of the GHS minimum weight spanning tree animation

- *Messages:*
  - *Accept:* This message is coloured light green. It shows the relation to message *reject*, but is still easy to distinguish from the dark green colour of *reject*.
  - *Connect:* This message is coloured red because of the idea that this edge will later be a tree edge of the MST and tree edges are always coloured red.
  - *Change-Root:* This message is coloured violet. As this message is sent after a sequence of blue *report* messages and will cause a red *connect* message, a colour between red and blue was selected.
  - *Initiate:* This message is coloured black because it is sent always along red tree edges such that a good contrast is guaranteed.
  - *Reject:* For this message a dark green was chosen as it shows the relation to accept and is distinguishable from the light green.
  - *Report:* The colour for this message is blue.
  - *Test:* This message is coloured in magenta.

Communication View

The communication view counts for each node all messages that have been sent by this node. It also shows the average number of messages that have been sent by a

process. Although the message size is constant, the message size is displayed below every process. For the initialization of this view the maximum possible number of messages is computed in order to guarantee a nice layout.

A process sends at most $\delta + \log n$ *test* messages because in every round a process sends *test* messages as long as there are non-tree edges which do not belong to the same component. The number of sent *accept* messages is in the worst case $\delta \log n$, while the number of *reject* messages is bounded by δ. If in every round a node has to send *initiate* to all its neighbours then additional $\delta \log n$ messages are sent. In each round one additional message for *change-root*, *connect* and *report* has to be considered. This results in a total cost of $\log n + \delta \log n + 2\delta + 2$ messages.

Causality View

The causality view appears as for all animations except from the colouring of the arrows. A relation between two processes is shown, when a message was sent from one process to the other. If a process receives a message it updates its local clock by taking $max(send\_tm, local\_tm) + 1$ as the new value, where $send\_tm$ denotes the local time of the sending process when the message was sent while $local\_tm$ denotes the current local time of the receiving process. An arrow will point from the row of the sending process at $send\_tm$ to the row of the receiving process at the new value of $local\_tm$. The colouring of the arrows was inspired by the idea that the user should be able to consider with each relation a component. Therefore the following colouring strategy was chosen:

- *initiate* and *report* messages are coloured always in the colour associated with the leader of the component. Note that with each node a colour is associated. This colour is the same for all views, but two nodes might have associated the same colour which results from the animation program having no previous knowledge about the number of processes.
- *test*, *reject*, *accept*, *change-root* and *connect* messages are coloured in the colour associated with the sender.

Process Step View

This view looks the same as in all other animations. A user can click on one node in the basic view and will see inside the Process Step View information about latest state, event, time and local clock. The user can also retrace a sequence of events by clicking inside this views on buttons “Previous Event” or “Next Event”.

Process Occupation View

The Process Occupation View shows in real time how long a process was busy with determining MWOEs or being leader of a component. In order to clarify the sequence of messages of kind *change-root* and *connect* which are used to determine a new leader, arrows are displayed for each such message. The arrow is coloured in the colour associated with the receiver pointing from senders row at sending time to receivers

row at receiving time. The bars showing the process occupation are formed in the same way as for other animation and coloured as the associated process is coloured.

Chapter 5

Resource Allocation

The heart of the *resource allocation problem* is about resolving conflicts between processes in a distributed system. Usually processes share common resources e.g. a shared memory where only one holder of the resource should be able to access the resource. Thus the problem can be described by a finite set of resources and a finite set of processes that compete for accessing their resources. Any solution of the problem is required to guarantee:

- *mutual exclusion:* no resource may be accessed by more than one process
- *no starvation:* as long as processes do not fail every process which is trying to access the resource will succeed in finite time.

In the following three algorithms are described which solve the *resource allocation* problem for an asynchronous message-passing network model. In section 5.1 the dining philosopher problem will give a formal description for the resource allocation problem. The algorithms introduced in section 5.5 and section 5.7 assume an initial graph colouring algorithm. Therefore in section 5.4 an algorithm using randomization is described which colours the graph with at most *degree of the graph* $+1$ different colours. The analysis uses the following convention: l denotes the upper bound given for the time that any process task needs to proceed and d denotes the upper bound given for the time that any message needs to traverse on a link i.e the time between sending event and receiving event of a message. Further n denotes the number of processes, D the diameter of the graph and δ the degree of the graph.

5.1 The General Dining Philosophers Problem

A formal description of the *resource allocation problem* for an asynchronous message passing system is given by the *dining philosophers problem*. It deals with a number of philosophers dining together. Each philosopher shares forks with other philosophers and is allowed to eat when possessing all shared forks. Again a solution to this problem must satisfy starvation freedom and mutual exclusion.

Let $G(V, E)$ be the communication graph of the modeled network, for which V denotes the set of processes and E the set of links between processes of V. Then,

every process $p \in V$ represents a philosopher, who works independently and may request resources at any time. Further for every resource which is shared between two processes p and q there exists a link $(p, q) \in E$ and vice versa. Thus, every process can request directly resources from its neighbours. A resource is represented by a token called fork. A process owns the resource if and only if it owns the fork. It can transmit its own fork to the neighbour with which it shares the resource. The fork will be sent via their common link.

It is assumed that every process can be in one of the following three states: *thinking*, *hungry* and *eating*. A process in state *thinking* is not interested in its neighbours resources and sends requested forks to its neighbours. A process will switch from state *thinking* to *hungry* when it wants to access its resources. Then it tries to collect all forks of its neighbours. When a process in state *hungry* collected all forks from its neighbours it will start accessing its resources. It changes to state *eating* in which it keeps all forks until it finishes with accessing the resources and changes to state *thinking* again.

The main problem is how to resolve starvation scenarios for two hungry processors competing for a resource. The following algorithms are solutions with different quality concerning time complexity and fault tolerance. It should be noted that that in the described case exactly two processors can share a resource, but the more general case where more than two processors share a resource can be extended from the above description.

5.2 The Ricart and Agrawala Algorithm

The algorithm of Ricart and Agrawala [13] resolves conflicts between processes by sending messages with time stamps. For this purpose a process which changes its state to *hungry* sends messages, called *request*, to all its neighbours. A *request* message includes information about the unique identifier of the sender and a time stamp. The time stamp is the local clock of the sending process when the message was created. It is increased when a message *request* is sent or received. Before sending a set of *request* messages the local clock is increased by one, while on the receiving of *request* the local clock will be set to

$$\textit{new value} := \max(\textit{old value}, \textit{received value}) + 1.$$

The time tuple $(\textit{local clock}, \textit{unique identifier})$ gives a lexical order of events. Although in real time the events might occurred in a different order the received lexical order of events results in an equivalent execution. Hence on the receiving of *request* a process, which is competing with another process for a resource, can distinguish whether receiver or sender were requesting first for a Resource assuming the lexical order of events.

Therefore, a process p that receives a message *request* from process q does the following depending on its state:

- If p is thinking, then it sends a message *fork* to q. Sending a *fork* message symbolizes that a process gives access to a resource and guarantees not to access the resource until it received itself a message *fork*.

- If p is hungry, then it is competing with q for the same resource. This means that p sent before a message *request* to q, so p compares by using the lexical order of events which process sent its message earlier. If p sent its message later then it replies by sending message *fork*. Otherwise it delays sending message *fork* until it finished with accessing its critical section.
- If p is eating it also delays sending message *fork* until it finished with its critical section.

A process may access its critical section, when it received a message *fork* from all its neighbours. When it is finished it sends to all neighbours which requested to access a resource messages *fork* and changes its state to *thinking*.

Correctness

The algorithm works correct if it guarantees mutual exclusion, progress and starvation freedom. These properties are shown by following lemmas.

Lemma 5.1 *The algorithm guarantees mutual exclusion.*

Proof. Let processes p and q be two arbitrary processes which share a resource. Assume that p and q were in their critical section at the time.

In order to gain access to their critical section processes p and q sent *request* messages to all their neighbours. In particular they sent *request* messages to each other. Let t_p denote the local clock time when process p sent its *request* messages and let t_q denote the local clock time when process q sent its *request* messages. Due to the lexical order of time either $t_p < t_q$ or $t_p > t_q$ is valid.

If $t_p < t_q$ was valid then p would have delayed replying the *request* message of process q until p finished accessing its critical section. This implies that $t_p > t_q$ was valid. However, if $t_p > t_q$ was valid q would have delayed replying the *request* message of p until q finished with accessing its critical section, leading to a contradiction that both processes were at the same time in the critical section. □

Lemma 5.2 *The algorithm guarantees progress.*

Proof. Assume there is a point in the execution in which some processes are hungry, and after this point no process can enter its critical section. Then the system will reach a state in which no messages are sent any longer and no process changes its state. Let p be a process in state *hungry* with smallest request time. All neighbours in state *thinking* must have answered the *request* messages of p with sending messages *fork*. As p is the process that sent its *request* messages with the smallest time tuple, also all processes in state *hungry* will send messages *fork* to p. Thus p will receive *fork* messages from all its neighbours and will change its state to *hungry*. This is a contradiction to the assumption that no progress happens. □

Lemma 5.3 *Every process is able to access its critical section.*

Proof. Receiving message *request* will make the local clock of a process at least one unit greater than the time stamp of message *request*. Therefore, a process which sends *request* messages to all its neighbours will loose at most once a competition with each neighbour while it is trying to access a resource. Assume some processes never manage to enter their critical section. Then, there must exist for these processes at least one neighbour which requested the shared resource before, and also cannot enter its critical section. Let $C := p_0, p_1, \ldots, p_l$ be the longest chain of neighbouring processes (p_i, p_{i+1}) such that all those processes never manage to enter their critical section and p_{i+1} prevents p_i from using its resource. Each process p_i requested the resources from its neighbours at local clock time t_i. Thus $t_0 > t_1 > \ldots > t_l$ must hold. All neighbours of p_l which requested earlier for their resource will succeed with entering their critical section since they are not part of C. Therefore, p_l will be able to enter its critical section because it will receive from all neighbours which sent their *request* messages earlier a message fork. Receiving all messages *fork* from neighbours that sent their messages later than p_l is guaranteed by the algorithm. This leads to a contradiction due to the assumption that p_l never manages to enter its critical section. □

Time Complexity

While a process is competing for a resource it will loose at most once a competition with each of its neighbours. The longest possible chain of neighbours that prevent each other from accessing a resource is n. Therefore a process has to wait in the worst case until all other $n - 1$ processes entered once their critical section and received all messages *fork*. This gives a time bound for accessing a resource of $O(n(l + d))$.

Message Complexity

In order to access the critical section a process needs to send at most d *request* messages and receives at most d *fork* messages. This gives a total communication complexity of $O(d)$ for each access of a critical section.

Fault Tolerance

A failure in the critical section of a process might stop all other processes from being able to access the critical section. Therefore, this algorithm does not guarantee any fault tolerance.

5.3 Animation of the Ricart and Agrawala Algorithm

The animation consists out of the views which were proposed in chapter 2. For each view a description of its usage is given in the following:

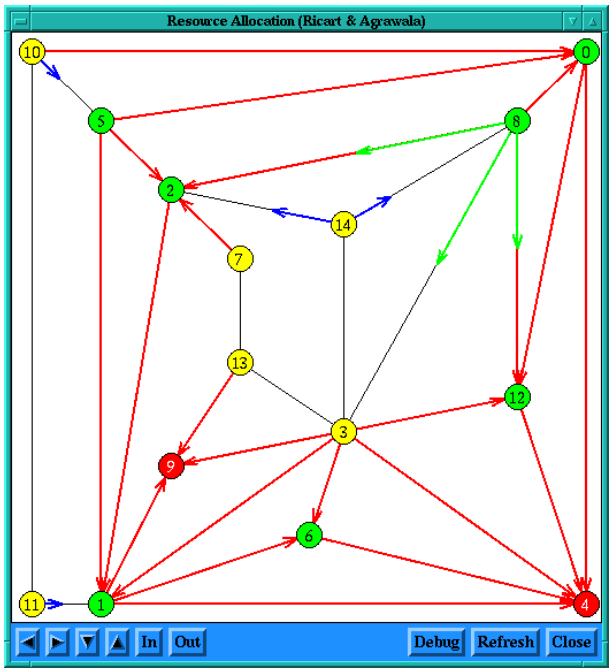

Figure 5.1: Basic view of the Ricart and Agrawala algorithm

Basic View

The basic view (cf. figure 5.1) shows the communication graph of the network in which each edge also represents a shared resource. Initially all nodes are shown as yellow circles. As the animation proceeds nodes change their colour to

- green when they get interested in a resource
- red when they managed to access their critical section
- yellow when they left their critical section

The unique identifiers used for the lexical ordering of events are displayed for each node. They also help to identify nodes in other views.

The edges appear as black coloured polylines by default. If two nodes are competing with each other for a resource then the edge changes into a red arrow. The arrow points to the node which may access a resource first.

For a message which is transmitted on a link an arrow will be shown until the message is received. The arrow points from sender to receiver and continuously changes its size along the edge connecting sender and receiver. A *request* message is coloured green, while a *fork* message is coloured blue. If both messages are sent at the same time one arrow will be shown which will flash between both colours.

Communication View

The communication view counts for each process the number of *request* messages which are necessary when trying to collect all neighbouring resources. When a process changes its state to *hungry* the number of sent messages will be set to zero and the process starts with counting *request* messages again. A mark shows the maximum number of messages which were necessary when trying to access all its resources. The average process shows the average for *request* messages sent by all processes for accessing once a resource. A mark shows the maximum average value during the execution. It will be easy to observe that the number of sent *request* messages is proportional to the degree of a process. Below each bar of a process the bit complexity of a message is shown. The bit complexity of messages is interesting for this algorithm as the user can observe the logarithmic increase of *request* messages. The bars and message sizes are coloured according to the colour associated with each process.

Causality View

The causal relations between processes are shown in form of arrows pointing from sender of a message to the receiver. It starts at the local clock time of the sender when sending the message and ends at local time of the receiving process when receiving the message. It should be noted that the local clock time for the causality view is computed different from the local clock used for the time stamp of *request* messages. On receiving a message the causality view's local clock is set to

$$\textit{new value} := \max(\textit{old value}, \textit{received value}) + 1.$$

The colouring of relations is realized by colouring all *request* messages in the associated colour of the sending process while *fork* messages uses the associated colour of the receiving process. This helps the user to see which relations were caused by each other.

Process Step View

The process step view has the same appearance as in all other animations. A user can click on one node in the basic view and will see inside the process step view information about latest state, event, time and local clock. The user can also retrace a sequence of events by clicking inside this view on buttons "Previous Event" or "Next Event" (see also page 8).

Process Occupation View

The process occupation view shows for each process a bar for the period of time when a process started to compete for its resources until the process left its critical section. In order to distinguish how long a process needed to access its resources and to spend inside its critical section, the bar is separated into two sections. The first section shows the period which a process needs to receive access to all its resources and is coloured in the associated colour of this process. The second section shows the access time inside the critical section and is coloured in a lighter shade of colour than the first part.

5.4 $\delta + 1$ Colouring by Luby

The $\delta + 1$ colouring problem is about associating a colour with each vertex of an undirected graph such that two neighbouring vertices of the graph receive different colours. A colouring algorithm can be used for the resource allocation problem by solving which kind of resources a process can access initially. Then, a process holds a fork when its determined colour is smaller then the colour of its neighbour and the longest chain of processes that wait for other processes with higher priority is reduced to $\delta + 1$ in the beginning. The algorithms in section 5.5 and section 5.7 will use the $\delta+1$ colouring in order to guarantee each process a quicker access to its critical section and gain a better fault tolerance.

The Algorithm

The following randomized algorithm is based on the algorithm by Luby described for a shared memory model [9] and implemented for a message passing system.

It is assumed that every vertex of the undirected graph is associated with a process and every edge with a link. Each process p initially knows the set of colours

$$avail(p) \leftarrow \{1, \ldots, deg(p) + 1\}$$

and the set of neighbours which have not decided for a colour

$$neighb(p) \leftarrow \{0, \ldots, deg(p) - 1\}.$$

The algorithm proceeds in phases and finishes when all processes determined a colour.

1. Process p starts with a new phase. With probability $\frac{1}{2}$, p chooses a random value $temp(p)$ out of the set $avail(p)$, otherwise it chooses $temp(p) := 0$. It sets $color(p) := temp(p)$ and sends a *colour* message including $color(p)$ to all neighbours in $neighb(p)$.

2. When p receives a *colour* message it compares the received value with $color(p)$. If these values are equal then $color(p)$ will be set to 0.

3. After receiving *colour* messages from all neighbours in $neighb(p)$, p will either send *confirm* messages to all neighbours if $color(p)$ holds or it will send *confirm* messages to all neighbours in $neighb(p)$. With each *confirm* message the current value $color(p)$ is sent.

4. When receiving a *confirm* message p will delete the sender from $neighb(p)$ if the value of the message is greater than 0. Then, it will also delete the value from the set $avail(p)$.

5. After receiving *confirm* messages from all neighbours $neighb(p)$ p will start with the next phase if $colour(p) = 0$ is determined.

6. Process p will terminate if $colour(p) > 0$ is determined and from all neighbours a *confirm* message with value greater than zero is received.

Correctness

The algorithm works correct if all processes can determine a colour which is different from all neighbouring processes.

Lemma 5.4 *For all processes p , $Pr(color(p) \neq 0) \geq \frac{1}{4}$ at the end of each phase.*

Proof. Fix an arbitrary process p. Let $t = |avail(p)|$. At the end of a phase

$$\begin{aligned}
&Pr(color(p) \neq 0) \\
&\quad = \sum_{c \in avail(p)} Pr(color(p) = c \mid temp(p) = c)\ Pr(temp(p) = c) \\
&\quad = \frac{1}{2t} \sum_{c \in avail(p)} (1 - Pr(color(p) = 0 \mid temp(p) = c) \\
&\quad = \frac{1}{2} - \frac{1}{2t} \sum_{c \in avail(p)} Pr(color(p) = 0 \mid temp(p) = c) \\
&\quad = \frac{1}{2} - \frac{1}{2t} Pr(\exists q \in neighb(p) : temp(q) = c \mid temp(p) = c) \\
&\quad \geq \frac{1}{2} - \frac{1}{2t} \sum_{c \in avail(p)} \sum_{q \in neighb(p)} Pr(temp(q) = c \mid temp(p) = c)
\end{aligned}$$

As q and p choose their values independently

$$\begin{aligned}
Pr(color(p) \neq 0) &\geq \frac{1}{2} - \frac{1}{2t} \sum_{c \in avail(p)} \sum_{q \in neighb(p)} Pr(temp(q) = c) \\
&= \frac{1}{2} - \frac{1}{2t} \sum_{q \in neighb(p)} \sum_{c \in avail(p)} Pr(temp(q) = c) \\
&\geq \frac{1}{2} - \frac{1}{2t} \sum_{q \in neighb(p)} \frac{1}{2}
\end{aligned}$$

and because of $|neighb(p)| \leq |avail(q)| = t$

$$Pr(color(p) \neq 0) \geq \frac{1}{4}$$

□

From lemma 5.4 it can be concluded that in each phase $\geq \frac{1}{4}$ processes are expected to decide for an colour. Hence all processes are expected to decide after $O(n \log n)$ phases. It remains to show that all neighbouring processes are of different colours.

Lemma 5.5 *After the end of the algorithm, all pairs of neighbours (p, q) have determined different colours.*

Proof. Assume for a process p the existence of $q \in neighb(p)$ such that $color(p) = color(q) > 0$. Then, p and q decided for the same value in the same phase. Otherwise

if p had determined a colour value earlier than q, p would have confirmed $color(p)$ such that the q would have removed $color(p)$ from $avail(q)$. Therefore, q could have never decided for a value which equals $color(p)$. For the same reason q would not have determined a colour value earlier than p.
As $color(p) = color(q)$ holds in phase k, also $temp(p) = temp(q)$ must be valid. If after the exchange of *colour* messages $temp(p) = temp(q)$ holds, processes p and q set $color(p) = color(q) = 0$ which leads to a contradiction. □

Time Complexity

In each phase a process which has not determined a colour will send *colour* messages to neighbours, receive *colour* messages from neighbours, send *confirm* messages to neighbours and receive confirm messages until it starts with the next phase or decides for a colour. This will take time $\leq 4(d + l)$ such that according to lemma 5.4 the algorithm is expected to terminate in time $O(n(d + l) \log n)$.

Communication Complexity

In each phase every process sends at most 2δ messages. Further $\geq \frac{1}{4}$ processes are expected to stop with sending messages at the end of a phase. Therefore an upper bound for the expected total amount of messages is given by

$$\begin{aligned} \delta n \sum_{k=0}^{\infty} \left(\frac{3}{4}\right)^k &\leq \delta n \frac{1}{1 - \frac{3}{4}} \\ &\leq \frac{4}{3}\delta n \end{aligned}$$

Hence a total of $O(\delta n)$ messages are expected until the algorithm terminates.

Animation

The purpose of this colouring algorithm is to give to a process a colour as an initial phase of a particular algorithm that needs this colour. Therefore the functionality of the colouring is of secondary interest and will only be symbolized by flashing the colour of a process as long as no colour could be determined.

5.5 The Chandy and Mistra Algorithm

The algorithm by Chandy and Mistra [2] resolves conflicts by defining for every possible conflict a *precedence*. When two processes compete for a resource the one with higher precedence may access the resource first. In order to receive a solution which is fair these precedences will have to change dynamically.

Chandy and Mistra derive from the undirected communication graph, in which edges represent shared resources between processes, a directed graph called *precedence graph*. For each resource an edge of the precedence graph is directed from processes with lower precedence to processes with higher precedence. The precedences of

the graph are chosen such that it is always possible to distinguish at least one process from all other processes i.e. this process can enter its critical section. This is ensured by the existence of at least one process which has higher precedence for all its shared resources. A process with this property is called *sink*. Its existence is guaranteed when the precedence graph is always acyclic. By changing directions of edges it is possible to change the precedences dynamically. This must happen in a way that the precedence graph stays acyclic, so *progress*, *fairness* and *mutual exclusion* is guaranteed.

Let H denote the precedence graph for a given communication graph. Then, it is enough to assume that each process has partial knowledge about H concerning the precedences to neighbour processes. Consequently, for the implementation of H Chandy and Mistra introduced *forks* which have the property to be either clean or dirty. A fork will be cleaned if and only if it is sent to a neighbour process. A clean fork will be dirty when it was used to eat i.e the holder of the resource entered the critical section. After use it remains dirty until it is sent to a neighbour process. The respective precedence graph H can be defined in the following way:

For all pairs of processes p and q which share a common resource,
$(p, q) \in H \Leftrightarrow$ one of the following statements is true:

1. p holds the fork for the resource and the fork is clean
2. q holds the fork for the resource and the fork is dirty
3. the fork for the resource is in transit from q to p

H is initialized acyclic for example by the colouring algorithm in section 5.4. It will remain acyclic if after use of the critical section a process reverse all adjacent precedences in one step. All edges are now directed from this process to neighbour processes and therefore no additional cycles occur.

The request of forks is realized by *request tokens*. For each fork there exist one request token such that only the holder of the request token can request a fork. A hungry process requests a fork by sending the request token to the owner of the desired fork. Then, a process is not interested in accessing its resources when it holds a request token but not a fork. Further a process which holds a request token and the respective fork has an outstanding request for a token.

The Algorithm

1. The algorithm is initialized by an acyclic precedence graph H and all processes with lower precedence own dirty forks while processes with higher precedence own request tokens. All processes are thinking i.e they are not interested in their resources.

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

3. A process which received all forks will change its state to *eating*.

4. A process which leaves the critical section changes the state of all its forks to dirty. Then for all held request token the respective fork is sent to neigbour processes.

The above steps assume following rules:

- *Receiving a request token for fork f:* If processors state is different from *eating* and f is dirty then f will be sent to the requeting processor. If processors state was also *hungry* then the request token will also be sent back.
- *Receiving a fork f:* The state of f will be set to clean.

Correctness

The algorithm is correct if mutual exclusion und starvation freedom i.e. fairness is guaranteed. The mutual exclusion property follows directly from the acyclic precedence graph. Thus it remains to show that H is always acyclic and every process manages to eat.

Lemma 5.6 *The precedence graph H is acyclic.*

Proof. Initially H is guaranteed to be acyclic by definition. An edge e of H will change the direction only if the accordant fork changes from clean to dirty (sending a fork will automaticly clean it such that the accordant edge keeps its direction). A process p will change the state of a fork if and only if it is leaving the criticical section. Then, it holds all forks shared with neighbour processes and changes the state of all forks to dirty. Hence, all indicent edges point away from p and therfore it cannot cause a cycle. □

Lemma 5.7 *Every process is able to enter the critical section.*

Proof. Let the depth in H of any process p be defined as the maximum number of edges along a path from p to another process without predecessor. The proof will show by induction that a process of depth k will eventually eat if predecessors at depth $k-1$ can eat.
k=0: Process p tries to access its resources and is of depth 0 i.e.there are no predecessors. For each fork which is needed by p one of the following is true:

1. The fork is clean and hold by p. Hence p will keep it until entering the critical section.
2. The fork is in transit from neighbour process q to p.
3. The fork is dirty and a neighbour process q holds the fork. Process q will receive a request token sent by p. Since q has lower precedence a request token is streight replied by sending the fork. The only exeption is when q is eating. Then, process q delays sending the fork until it left the critical section.

Therefore p will receive latest after time $2(d+l)$ all forks and manages to eat.

k-1 $\rightarrow$ k: Assume all processes at depth $k-1$ will manage to eat and process p is of depth k when it tries to access its resources. Each fork f which is tried to be collected by p knows the states dirty or clean.

case f is clean: If f is hold by p it will remain by p until leaving the critical section. Oherwise, it is held by a neighbour q with higher precedence. Process q will manage to eat and will have to give p higher precedence to the fork. Hence on a request the fork will be sent to p and remain there until p can eat.

case f is dirty: If f is hold by p it might be requested by a process with a higher precedence and changes its state to clean and similar to the above case p can receive f in state clean. Otherwise p has higher precedence and will succeed with requesting a fork latest in time $2(d+l)$ (similar to the case $k=0$).

Therefore p will access in finite time all forks and manages to eat. □

Time Complexity

From the correctness proof it can be concluded that a process at depth k will have to wait for forks from neighbours with lower precedence time $2(d+l)$ plus the time neighbours of depth $\leq k-1$ needs to be able to eat. This results in following recursion:

$$\begin{aligned} T(k) &= T(k-1) + 4(d+l) \\ &\leq 4(k+1)(d+l) \end{aligned}$$

Although the acyclic graph is initially of depth δ, the worst case depth is n due to transformations of H. Therefore, a process must wait time $O(n(d+l))$.

Communication Complexity

A process sends at most one request token to each neighbour and receives from each neighbour at most one fork. Hence $O(\delta)$ messages are necessary until a process can access the critical section.

The size of messages is always constant.

Fault Tolerance

If a process at depth 0 fails processes at depth n might never get access to their critical section. Hence this algorithm does not guarantee any fault tolerance.

Conclusion

The only real improvement to the algorithm by Ricart and Agrawala is the constant size of messages while message complexity and time complexity stays the same. Nevertheless one would expect better results by the algorithm of Chandy Mistra when the number of conflicts is low. Then, a process might not need to request all forks because it might still hold some dirty forks. The algorithm of Ricart and Agrawala assumes

sending requests to all neighbours. Hence a lower amount of traffic and shorter access time to the critical section could be expected by the algorithm of Chandy and Mistra.

5.6 Animation of the Chandy and Mistra Algorithm

The animation consists out of two parts. The introductory part shows how processes determine an acyclic graph by performing the colouring algorithm by Luby(see section 5.4). The main part shows the algorithm by Chandy Mistra. Due to asynchrony both parts are not necessarily separated from each other. The user might observe that some processes already try to access their resources while other still try to determine a colour.

The animation is built according to chapter 2. Except from the basic view all other views are related only to the second part of the animation. A detailed description for each view is given in the following:

Basic View

Initially the basic view (cf. figure 5.2) shows the communication graph of the network. All processes appear in the shape of yellow circles while links are shown as black polylines. When processes wake up they begin with the colouring phase. As long as a process has not decided for a colour it is continuously flashing colours between a dark and a light blue. When a colour is determined the colour of a process will change to yellow again.

The colouring phase for a process p is finished when all neighbour colours are received. Hence p will initialize the partial knowledge of the precedence graph by taking a fork shared with neighbour q if colour of p is smaller than colour of q and q did not request this fork before. Otherwise p will take a request token. This is visualized by showing an red coloured arrow which points to the holder of a fork. The arrow appears dotted if the fork is dirty (which is initially the case) or solid if a clean fork is hold.

In the main phase of the animation processes are coloured according to their states:

- A process will be coloured yellow when it is not interested in accessing the critical section.
- If a process is trying to access all its resources it will be coloured green.
- A red colour will be applied if a process is inside the critical section.

Similar to the initialization for each edge a red solid or dotted arrow will indicate which node is the holder of the fork. If a fork is in transit from one process to another this will be shown by a solid arrow which is continuously resizing its length. Sending a request token is indicated by a blue arrow continuously resizing its length. It should be noted that a fork and a request token can be sent to a process on the same link at the same time. The user will recognize this by observing a resizing arrow which is flashing between red and blue.

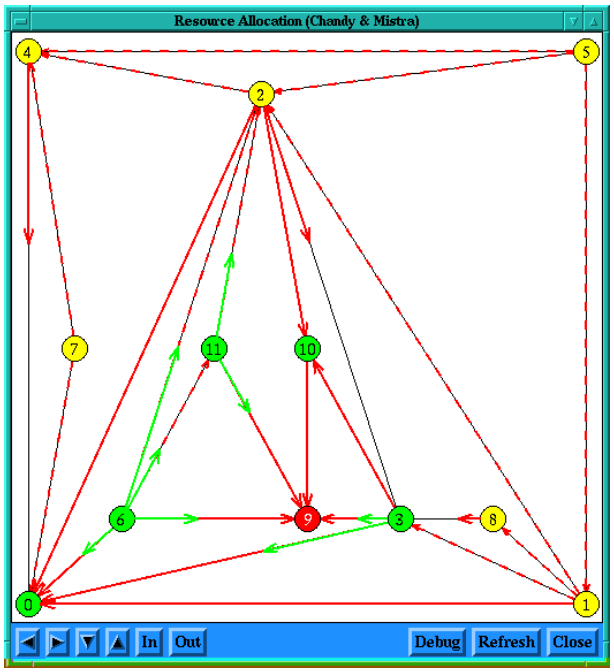

Figure 5.2: Basic view of the Chandy and Mistra algorithm

Communication View

Analogous to the animation of Ricart and Agrawala (see section 5.3) the communication view counts the number of request tokens which a process needed to send in order to gain access to the critical section. When process p becomes interested in its resources the number of sent request tokens will be initialized with zero. For each sent request token p's associated bar increases by one unit. A mark will indicate the maximum number of send request tokens among all attempts of entering once the critical section. The average bar shows the average number of sent request tokens for accessing once the critical section. A mark will show the highest average value. All bars are coloured according to the colours associated with each process.

The user will observe that the number of sent request token might strongly differ between two attempts of accessing resources. However, the shown marks will indicate that the maximum number of send messages is proportional to the degree of a process.

Causality View

The causality view shows causal relations given by send and receive events considering the local time when events occurred. An arrow will point from sender to receiver starting at local time of the sender and ending at local time of the receiver. The local clock time of a process will be changed when a message is received. Then, the local clock will be set to

$$\textit{new value} := \max(\textit{old value}, \textit{received value}) + 1.$$

The colouring of a relation depends on the occurred event. If a request token is sent the animation colours the accordant relation by using the colour associated with the sending process. If a fork message is sent the animation uses the colour associated with the receiver.

Process Step View

As in all other animations a user can click on a process inside the basic view in order to see inside the process step view information about latest state, event, time and local clock. The user can also retrace a sequence of events by clicking inside this view on buttons “Previous Event” or “Next Event” (see also page 8).

Process Occupation View

Similar to the animation of Ricart and Agrawala (see section 5.3) the process occupation view shows for each process the periods of time between switching to state *hungry* and finisth eating. Each time period is indicated by a bar. Since a user might also desire to distinguish between the period of trying and eating, the animation separates the bar into two parts. The first part will be shown in the associated colour of a process when it tries to access resources. The second part will use the same colour in a lighter shade indicating that a processor eats.

5.7 The Choy and Singh Algorithms

The algorithms by Choy and Singh [3] are motivated by the aim of achieving a better time complexity and fault tolerance than previous algorithms could guarantee. Hereby the idea of using a $\delta + 1$-coloring conflict solution should reduce the maximum access time. As described for the algorithm by Chandy and Mistra (see section 5.5) a $\delta + 1$-coloring will result in an acyclic precedence graph. In contrast to Chandy and Mistra the precedences will remain static and hence reduce the maximum chain of processes waiting for resources of neighbours with higher precedence to δ. The problem that a static precedence solution could hinder some processes entering their critical section is avoided by a mechanism called double doorway. This will guarantee that processes with lower precedence will loose at most once a competition with each higher neighbour and consequently will not starve.

Doorways

A doorway is a separation mechanism between two areas e.g. rooms for managing which processes are allowed to enter an area. Processes which passed a doorway at time t will prevent neighbour processes entering the same area at greater time than t until exiting the doorway. Thus a doorway guarantees for a process that after a certain point in time no neighbours will access an area. However, it is still possible that in spite of using a doorway neighbour processes will occur in an area, since moving from one area to another will take some time.

Asynchronous Doorway

An asynchronous doorway is implemented by requiring for each process p which desires to enter the doorway to check neighbours states. Let N_p denote the set of neighbours of process p and L_{pq} the actual state of q known by p. If neighbour processes $q_0, \ldots, q_l \in N_p$ have entered the doorway, p must wait until $q_0, \ldots, q_l$ will have exited the doorway. The code for entering and exiting this doorway can be described as it is shown in figure 5.3.

Doorway entry code:
($\forall q \in N_p$: **wait until** $L_{pq} \neq m_1$);
broadcast message m_1 to neighbours;

Doorway exit code:
broadcast a message different from m_1 to neighbours;

Figure 5.3: Asynchronous doorway entry code

Note that process q_i, $0 \leq i \leq l$, will not block p after leaving the doorway even if q_i would try straight after leaving to enter the same doorway again. In the worst case q_i and p would both enter the doorway at the same time since p considers only processes which passed the doorway before it started trying. For the same reason any other neighbour process r cannot prevent p from passing the doorway when r passed the doorway after p checked the state of its neighbours.

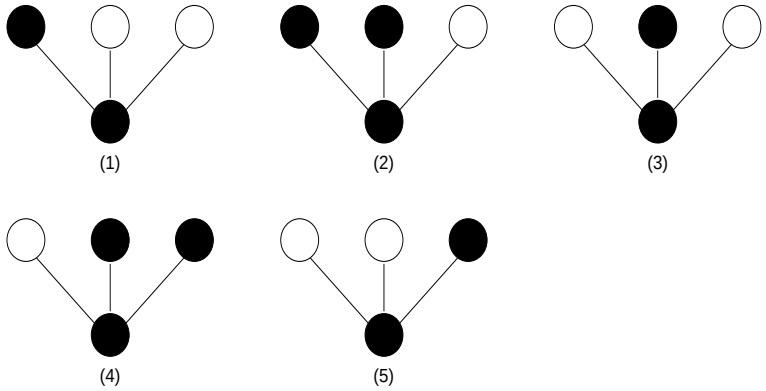

Figure 5.4: Example of a process being successively blocked in the doorway by its neighbours. The filled circles denote processes which have entered the doorway. Note that processes which enter the doorway in (2) and (4) must have been waiting for some other processes leaving the doorway.

The attempt of using such a doorway together with a $\delta + 1$-colouring algorithm for solving the resource allocation problem would lead to a solution where processes might have to wait time which is exponentially in δ. The reason is given by the possibility that a process p might successively be blocked by lower coloured neighbour processes after p entered the doorway (see example of blocked processes in figure 5.4). Recursively a lower coloured neighbour q of p, can be blocked successively by q's lower coloured neighbours. Let $T(c)$ denote the maximum time which a process of colour c will have to wait. Assume every process with colour > 0 has exact $\delta - 1$ neighbour processes with lower colour. Further let us assume an execution where each

process will get successively blocked by its lower coloured processes. Then a process with colour c will need time

$$T(c) = (\delta - 1)T(c - 1) = (\delta - 1)^c$$

and therefore

$$T(\delta) = (\delta - 1)^\delta$$

until a process will be able to access its critical section.

Synchronous Doorway

A process which desires to enter a synchronous doorway is required to wait for a situation in which all neighbours are outside the doorway. This is implemented by a process checking states of neighbours before entering the doorway. If all states show that no other process has entered the doorway the process will be able to enter itself. The code for entering and exiting this doorway can be described as it is shown in figure 5.5.

Doorway entry code:
wait until $(\forall q \in N_p: L_{pq} \neq m_2)$;
broadcast message m_2 to neighbours;

Doorway exit code:
broadcast a message different from m_2 to neighbours;

Figure 5.5: Synchronous doorway entry code

Note that it is still possible that more than one processes passed the doorway at the same time since all processes found their neighbours outside the doorway and therefore were allowed to pass the doorway.

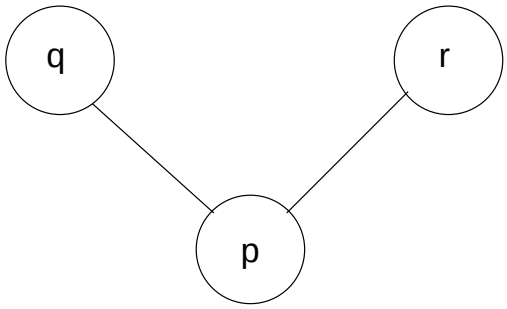

Figure 5.6: Example of a graph structure where two processes q, r can hinder process p from entering a synchronous doorway when either p or q will be inside the doorway.

Although a synchronous doorway allows no process to be successively blocked inside a doorway it does not solve the dining philosopher problem fairly because some processes might never be able to enter the doorway. Assume three processes p, q and r using the graph structure of figure 5.6. It is possible that q and r will cooperate such that at least one of them will be inside the doorway. Therefore process q waits until r is

inside the doorway before exiting (this is possible since q and r are no neighbours and thus will not prevent each other passing the doorway). and r will do the same before it is exiting the doorway. Hence p will never be able to gain excess to the doorway.

Double Doorway

By using the previous described doorways for resource allocation either processes could be blocked before entering the doorway (synchronous case) or while inside the doorway (asynchronous case). On the other hand the usage of a static precedence graph will need some further selection criteria for processes like even doorways. The double doorway, a combination out of the previous described doorways, will prevent processes from being blocked for long periods inside or outside the doorway and hence lead to a solution of the resource allocation problem in which higher fault tolerance and faster access time to the critical section can be achieved.

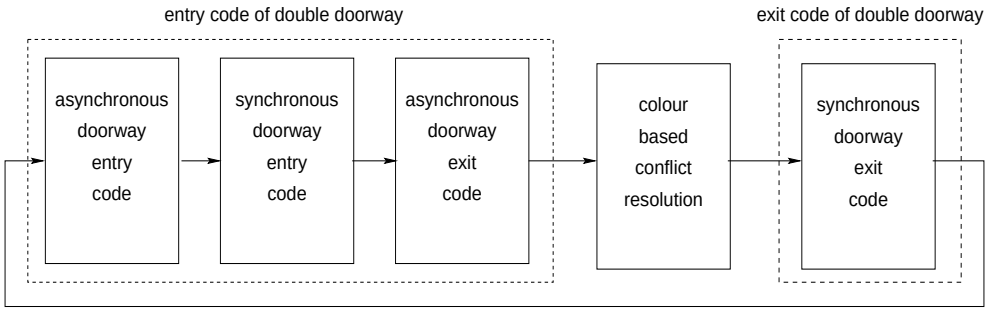

Figure 5.7: A double doorway together with a static colour based conflict solution.

As shown in figure 5.7 the entry code consists out of the synchronous doorway entry code embedded in an asynchronous doorway entry and exit code. Processes that wish to enter the double doorway cannot forever be hindered from entering by neighbour processes as the asynchronous doorway will allow a process to proceed latest when all neighbours inside the doorway have passed the asynchronous doorway exit code. Further the asynchronous doorway guarantees that process p which passed the asynchronous doorway will be blocked from entering the synchronous doorway at most once by each neighbour. If a process q could block p once more after passing the synchronous doorway it would try to enter the asynchronous doorway while p was still trying to enter the synchronous doorway. This leads to a contradiction to the property of the asynchronous doorway which will disallow q to pass until q passed the asynchronous doorway exit code. Also the problem of successive blocking after passing the doorway will be avoided by the double doorway using the synchronous doorway entry code. Hence no neighbour process can pass the double doorway after a process is known to be inside the doorway and thus neighbour processes cannot successively block this process.

5.7.1 A Solution with Failure Locality δ

From the previous described double doorway one could gain directly an algorithm for solving the resource allocation problem. However, the double doorway can be optimized concerning which neighbours are blocked at which kind of doorway and therefore save unnecessary blocking of processes and messages by the implementation. In the synchronous doorway one wishes to avoid that higher coloured processes will not starve because of lower coloured processes when using the static precedence graph. On the other hand the asynchronous doorway will guarantee that lower coloured processes will not forever be blocked by higher coloured processes passing the synchronous doorway. As a result the synchronous doorway will block only lower coloured neighbours while the asynchronous doorway will block only higher coloured neighbours.

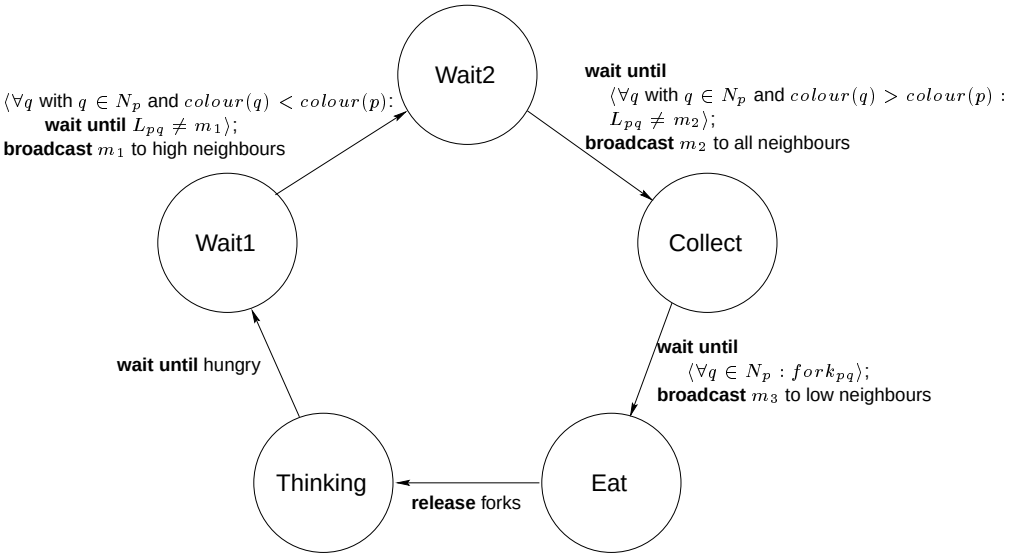

Figure 5.8: The state diagram of an algorithm with fault tolerance δ

The state diagram of figure 5.8 shows a solution in which the optimized double doorway is used. The solution uses states *thinking* and *eat* as in previous versions. A process in state *thinking* is not interested in its resources and will not compete with other processes until it wants to access its critical section, while a process in state *eat* has access to all its resources and is inside its critical section. Due to the double doorway the *hungry* state of the standard resource allocation solution is split in three states:

- In state *wait1* a process waits for its lower coloured neighbours inside the asynchronous doorway (in state *wait2*) to exit the asynchronous doorway (meaning neighbours change their state to *collect*).
- If a process is in state *wait2* a process will wait until no higher coloured neighbours are inside the double doorway (no higher coloured neighbour is in state *collect*). Process p of state *wait2* blocks higher coloured neighbour processes from entering the asynchronous doorway because p sent message m_1 to all those neighbours before it had changed its state from *wait1* to *wait2*.

- In state *collect* process p tries to receive access to its resources. By sending message m_2 to all neighbours before entering state *collect* p ensures that on the one hand all processes know about p requesting its forks and on the other hand lower coloured processes will be blocked from passing the synchronous doorway. When p in state *collect* received all forks it signals all lower coloured neighbours by sending m_3 that they are not blocked any longer and p starts eating.

When process p receives a message m_i by neighbour q, it will update its local state variable $L_{pq} := m_i$. Since message m_2 also equals a *request* message, p in state *thinking, wait1* or *wait2* would also send a message *fork* to q if $fork_{pq} = 1$. In state *collect* p will only send message *fork* if $fork_{pq} = 1$ and $colour(p) > colour(q)$ holds. When p is in state *eat* it will send messages *fork* to neighbour $q \in N_p$ after leaving its critical section if $fork_{pq} = 1$ and $L_{pq} = m_2$ is valid. A process p which sends a fork to neighbour q sets $fork_{pq} = 0$.

Correctness

The algorithm is correct if it satisfies mutual exclusion and fairness which will be shown by lemma 5.8 and lemma 5.10.

Lemma 5.8 *The algorithm of Choy and Singh guarantees mutual exclusion.*

Proof. With every resource a unique fork is associated. As a process will enter its critical section only if it possesses all forks which are shared with neighbours, none of the neighbour processes will enter the critical section at the same time. Also a process inside its critical section will keep all forks until it finished with eating. Thus mutual exclusion is guaranteed. □

Lemma 5.9 will show that any process in state *collect* will reach in a finite number of steps the critical section. Using this result lemma 5.10 will show, that a process in any state finally will succeed with entering its critical section.

Lemma 5.9 *Processes which managed to enter state* collect *will manage to eat in time* $O(\delta(d + l))$.

Proof. A process that changed to state *collect* will send message m_2 to all lower coloured neighbours in order to signal them not to enter state *collect*. Hence, a process has at most $(d+l)$ time units to change to state *collect* after a higher coloured neighbour sent m_2. Thus, process p of state *collect* and colour c can be sure that after time $c(d+l)$ for any path $(p = q_0, \ldots, q_l)$, in which q_{i+1} is lower coloured neighbour of q_i in state *collect*, no other lower coloured neighbour in $\{q_0, \ldots, q_l\}$ will reach state *collect*. The longest of those paths will decrease by one latest after time $2(d + l)$ since the lowest coloured processes will directly manage to access their resources after requesting them and processes inside the critical section will transmit forks after time $(d + l)$. By induction p will need time $(2c + 1)(d + l)$ until it can access the critical section. As $c \leq \delta$ holds a time bound of $O(\delta(d + l))$ can be concluded. □

Lemma 5.10 *The algorithm of Choy and Singh guarantees starvation freedom.*

Proof. A process in state *wait1* could be prevented to proceed to state *wait2* only by lowered coloured neighbours in state *wait2*. Further a process in state *wait2* would only be hindered by processes in state *collect* of higher colour. Processes hindering lower coloured neighbours to proceed to state *collect* will not be able to enter state *wait2* until the blocked lower coloured processes will have managed to proceed to state *collect* due to the asynchronous doorway. Hence, if a process of state *collect* will mange to eat all other processes will manage to eat and no process can starve. This was the result of lemma 5.9. □

Time Complexity

From lemma 5.9 it is known that a process in state *collect* will need time $O(\delta(d+l))$ until it will manage to eat, but it is not shown how long a process will need to pass the double doorway.

Lemma 5.11 *A process in state* wait2 *will need time* $O(\delta^2(d+l))$ *to transit to state* collect.

Proof. A process p which enters state *wait2* will send message m_1 to all higher coloured neighbours. It will take time $(d+l)$ until all higher coloured neighbours know about p's state and stop entering state *wait2*. Therefore the last higher coloured neighbour could enter state *wait2* $2(d+l)$ time units after p had sent $m1$. Process p can proceed latest when all higher coloured neighbours in state *collect* or *wait2* have managed to eat. Higher coloured neighbour processes of state *collect* will eat latest after time $O(\delta(d+l))$, while higher coloured neighbours might have to wait for their higher coloured neighbours to eat. By recursion one gains for a process of colour c following upper bound to access the critical section:

$$\begin{aligned} T(c) &\leq 2(d+l) + const\delta(d+l) + T(c+1) \\ &= (\delta - c)(2(d+l) + const\delta(d+l)) \\ &= O(\delta^2(d+l)) \end{aligned}$$

□

Lemma 5.12 *A process in state* wait1 *will manage to enter state* wait2 *in time* $O(\delta^2(d+l))$.

Proof. Due to the asynchronous doorway this time results directly from lemma 5.11. Let process p be in state *wait1* and $\{q_0, \ldots, q_l\}$ denote the set of lower coloured neighbours known to be in state *wait2* when p got hungry. Process p is allowed to proceed to state *wait2* when $q_0, \ldots, q_l$ managed to proceed to state *collect*. According to lemma 5.11 this will happen after time $O(\delta^2(d+l))$. □

Therefore a process is guaranteed to enter the critical section in time

$$O(\delta^2(d+l)) + O(\delta^2(d+l)) + O(\delta(d+l)) = O(\delta^2(d+l)).$$

Communication Complexity

In order to access all resources a process sends m_1, m_2 to all higher coloured neighbours, while m_2 and m_3 is sent to all lower neighbours. Hence it needs to send $O(\delta)$ messages.

Fault Tolerance

The fault tolerance of process p depends on the distance of the furthest neighbour process which is not allowed to fail. The analysis will show for a process which fails the furthest possible distance in which processes might be affected. Processes may fail in different states, but in the worst case a process fails when it has collected all forks. This could happen in any state as a process will release only requested forks. Neighbour processes will not be able to proceed further than state *collect* since they will never be able to collect all forks and thus never manage to eat.

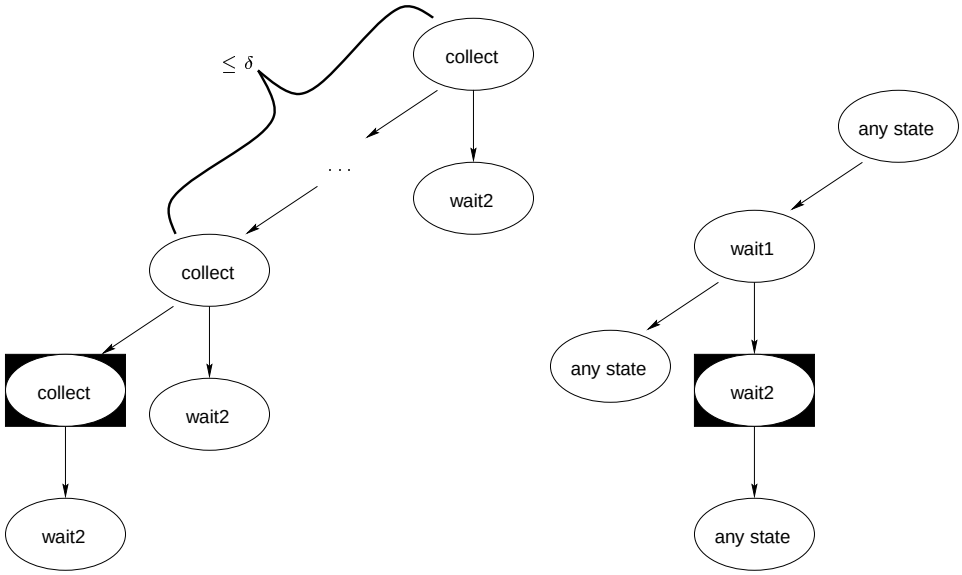

Figure 5.9: This diagram shows how the neighbourhood is affected when a process remains forever in state *collect* or *wait2*. The arrows show precedences between processes i.e. whether the neighbour process uses a higher or lower colour.

As shown in figure 5.9 a process that stays forever in state *collect* will force its lower and higher neighbours to remain in certain states. Straight from the description of the doorways it follows that

- processes with higher colour will also remain forever in state *collect* if they reach this state.
- processes with lower colour will remain forever in state *wait2* if they reach this state

Notifying figure 5.9 it can be concluded that a process will not starve if none of its neighbours remain forever in state *collect* or *wait2* and neighbours do not fail. Further

a failing process will affect other processes to remain in state *collect* forever only if they are not further distant than $\delta + 1$ because only higher coloured neighbours are affected in this sense and the maximum colour is limited by δ. Hence, a process will affect processes to remain forever in state *wait2* only if they are not further distant than $\delta + 2$ since a processes will remain in state *wait2* only if a neighbour process remains forever in state *collect*.

Therefore, a process will manage to eat if none process in the neighbourhood of $\delta + 4$ fails. The algorithm guarantees fault tolerance $O(\delta)$.

5.7.2 A Solution with Failure Locality 4

In the previous algorithm of 5.7.1 processes would never manage to collect all forks if lower coloured neighbours of state *collect* could not eat. This allowed chains with δ processes in state *collect* waiting for the forks of their lower coloured neighbours. In order to remove such kind of chains a new collection scheme is used for the following algorithm. Although the algorithm will guarantee a better fault tolerance than the previous of 5.7.1, a process will need to send more messages in order to achieve access to its critical section.

A New Fork Collection Scheme

When a process p starts with collecting its forks it will proceed in the following way:

1. Process p requests forks only from lower coloured neighbours and waits until all requested forks are received. During this time when p receives a *request* message for one of its forks it will release the fork.

2. If p received all forks from lower neighbours, it starts with requesting forks from all higher coloured neighbours. When p receives a *request* message from a higher coloured neighbour, it will delay answering until

   - either p finished with eating
   - or p has to release its forks because a lower coloured neighbour requested one of its forks.

3. When p receives a *request* message from a lower coloured neighbour it will release all requested forks and continue with 1.

4. If p received all forks it will eat and release all forks after eating.

The algorithm

Apart from the collection scheme the new algorithm works exactly as the algorithm of 5.7.1 which is shown in figure 5.8. Receiving message m_2 will not be interpreted as a *request* message and does not lead to releasing forks. For this purpose the algorithm sends extra *request* messages in state *collect* according to the previous described collection scheme. Assume process p receives a *request* message by q:

- if p is in state *thinking, wait1, wait2* it will reply with sending message *fork* and updating its value $fork_{pq} = 0$
- if p is in state *collect* and has lower precedence($colour(p) > colour(q)$) it will reply with sending message *fork* and updating its value $fork_{pq} = 0$
- p will also reply with sending message *fork* and updating its value $fork_{pq} = 0$ if p has not collected all forks from lower coloured neighbours
- if p of state *collect* has higher precedence than q and also collected all lower coloured forks it will delay sending message *fork* until it either will manage to eat or it will have to release a lower coloured fork before eating
- if p is eating it will delay releasing the fork until exiting the critical section.

Correctness

The mutual exclusion property is still valid for the same reason as given in the proof of lemma 5.8. Although the collection scheme has changed a process which entered state *collect* is still guaranteed to enter its critical section in time $O(\delta(d + l))$. Considering that processes will receive requested forks from higher coloured neighbours after time $2(d+l)$ the proof of lemma 5.9 will explain why. Moreover the new collection scheme does not affect processes in other states than *collect*. Therefore, it can be concluded from lemma 5.10 that the algorithm is still starvation free.

Time Complexity

As the time bound of lemma 5.8 still holds the time complexity of $O(\delta^2(d + l))$ will not change either.

Message Complexity

A process will have to send 2δ messages of kind m_i, $i \in \{1, 2, 3\}$ because it sends m_1 and m_2 to all higher coloured neighbours, while it sends m_2 and m_3 to all lower coloured neighbours. Further a process will have to request forks from all its neighbours. Due to the new collection scheme a process must release its forks whenever a lower coloured neighbour request a fork. This could lead to an exponential size of messages, if the period until a process will be able to eat was not limited by $k\delta(d + l)$, where k denotes a constant. Let s denote the lower bound which a process needs to perform any process task and let v denote the lower bound which a message needs to traverse on a link. Then a process will be able to receive a requested fork from a neighbour in time $\geq 2(s + v)$. In the worst case a process has to release forks straight after receiving them. A process only send *request* messages after releasing a fork. Hence, for each neighbour a process sends at most $\frac{k\delta(d+l)}{2(s+v)}$ messages until it collected all forks and thus $\frac{k(d+l)}{2(s+v)}\delta^2$ to all neighbours.

Altogether this results in a total amount of $O(\delta^2)$ messages which a process can send in order to achieve access to its critical section. The size of messages will remain constant.

Fault Tolerance

Similar to the previous algorithm in 5.7.1 the analysis will show the maximum possible distance of affected processes if a process fails. A process p of state collect will not receive a requested fork from a neighbour process q because either

- q failed
- q is inside its critical section
- q is lower coloured neighbour of p and has collected all forks from lower coloured neighbours, but is still waiting for higher coloured neighbours to send their forks.

If some process fails, neighbour processes will not be able to collect all forks. A neighbour process p of a failing process which enters state *collect* will not be able to proceed any further and thus will remain forever in state *collect*. Although p will never be able to collect all forks, it could manage to collect at least all lower coloured forks (only if the failing process was of higher colour) and for this reason cause a higher coloured neighbour process q to remain forever in state *collect*. Notifying that q will never be able to get access to all its lower coloured forks (because of p), neighbours r of q behave in the following way:

- If $colour(r_i) < colour(q)$ r_i will remain forever in state *wait2* when entering this state.
- If $colour(r_i) > colour(q)$ r_i will be able to reach state *collect* and also receive a requested fork by q. Hence it will manage to eat in time $O(\delta(d + l))$.

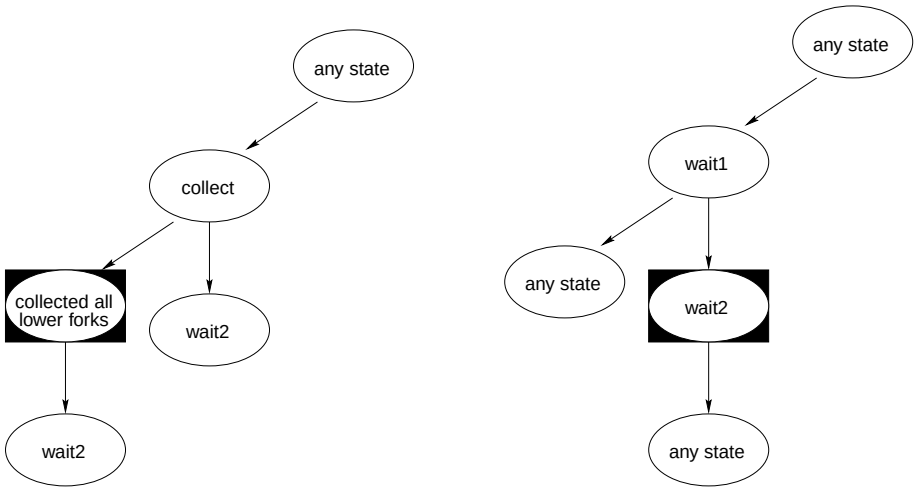

Figure 5.10: This diagram shows how the neighbourhood is affected when a process remains forever in state *collect* or *wait2*. The arrows show precedences between processes i.e. whether the neighbour process uses a higher or lower colour.

The analysis of a process remaining forever in state *wait2* is the same as described for the algorithm in 5.7.1 and shown in the state diagram of figure 5.10. One can see that a process is guaranteed not to starve when in the neighbourhood of no processes

remain forever in state *wait2*. This holds if no processes in the neighbourhood of 2 remain forever in state *collect* and hence processes may not fail in the neighbourhood of 4. The algorithm guarantees fault tolerance of 4.

5.8 Animation of the Choy and Singh Algorithms

This section describes the animations visualizing the algorithms in 5.7.1 and 5.7.2. Since the two algorithms differ only in the fork collection scheme their animation is constructed quite similar and for this reason only one description for both algorithm is given. Hereby the animations were built according to the general considerations of chapter 2, as this was also the case for previous animations.

The algorithms are initialized by Luby's colouring algorithm in 5.4 in order to achieve a $\delta + 1$ colouring of the graph. The colouring phase is the first part of the animation affecting only the basic view. The second and main part starts with the algorithms by Choy and Singh. Due to asynchrony some processes might still perform the colouring algorithm while others have already started with the main part.

In the following a description will be given for each view. It should be notified that the basic view for these algorithms consists out of two windows due to the complexity of the algorithm. The *main window* shows the communication graph and messages transmitted among the system, while the *state window*, informs the user about states of processes. These views will be explained and shown separately.

Basic View: Main Window

The main window (cf. figure 5.11) shows the communication graph and messages transmitted among the system as usually the basic view does. Inside the main window processes will appear as yellow circles while links are shown as black polylines. Initially processes do not have knowledge about the precedences towards neighbour processes. Hence the polylines representing links will be shown undirected. After waking up, processes start continuously flashing between a dark and light blue visualizing the colouring phase of the algorithm. When a process determines a colour the accordant circle will appear in yellow colour again. If also neighbour processes determined a colour the precedences are shown by transforming the undirected polyline into a directed polyline whose arrow points to the process with lower colour. The process with lower colour also takes the fork associated with the link between both processes. This will be represented by a blue dotted polyarrow pointing to the owner of the fork. The fork appears dotted since the user should also be able to see the colour of the underlying edge.

In the main phase of the algorithm processes are coloured according to their state. A process will appear

- yellow if it is in state *thinking*.
- green when it is hungry. The green colour will be shaded
  - light when the process cycles in state *wait1*.

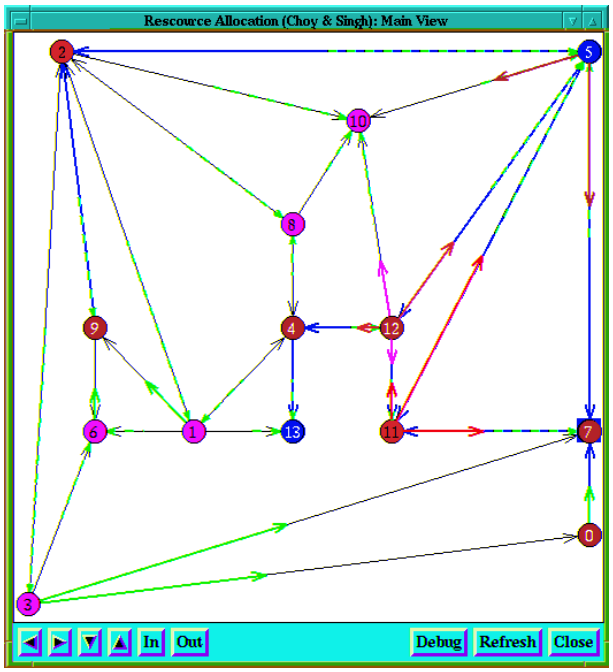

Figure 5.11: Basic view of the Choy and Singh algorithm

  - medium when the process cycles in state *wait2*.
  - dark when the process passed all doorways and starts with collecting forks.

- red when it eats.

The algorithm with fault tolerance 4 also distinguishes whether a process could collect all lower coloured neighbours. This is shown by a red mark around a process.

As mentioned before edges will appear after the colouring phase as directed black polylines. If two neighbour processes compete with each other for a resource the accordant edge will change its colour to red until one process finished with eating. This should help to observe chains caused by competitions.

While a message is transmitted on a link an arrow will be shown. The arrow points from sender to receiver and continuously changes its size along the edge connecting sender and receiver. For each message a different colour is used. When there are more than one messages in transit this will cause the arrow flashing between the colours of sent messages. In the following the colours associated with each message are listed:

- m_1: This message is sent when a process passed the asynchronous doorway and switched from state *wait1* to *wait2*. The message will be coloured in a light shaded green which is the colour processes have when they stay outside the asynchronous doorway because of having received m_1.

- m_2: Message m_2 is sent when a process passed the synchronous doorway and entered the state *collect*. The message will be coloured in a medium shaded

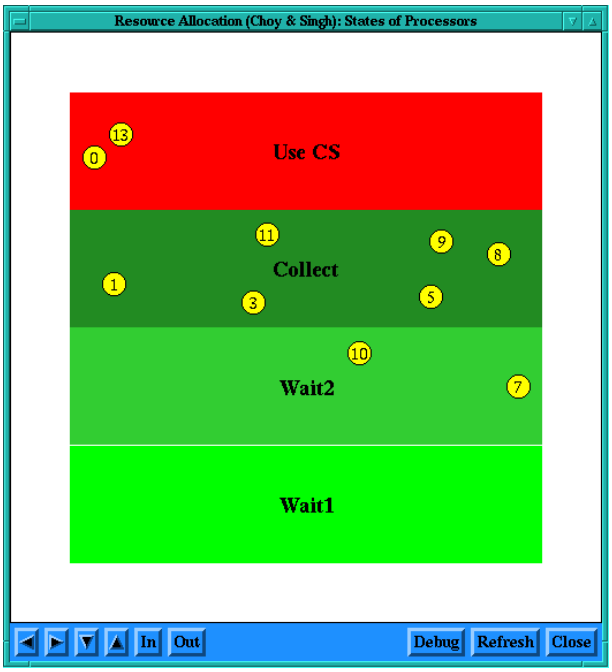

Figure 5.12: State window of the Choy and Singh algorithm

green which is the colour processes have which will have to remain outside the synchronous doorway because of having received m_2.

- m_3: Message m_3 is sent when a process has collected all forks and can start to eat. Neighbour processes are signaled that they are not any longer blocked by this process to enter state *collect*. Therefore m_8 is coloured in a dark shaded green according to processes in state *collect*.

- *fork*: According to the initialization of the algorithm forks are represented by a dotted blue arrow pointing to the owner of the fork. Hence, m_8 is also coloured blue.

- *request*: This message is only needed the 4-fault-tolerant algorithm to request forks from neighbour processes. Message *fork* is shown in a yellow colour.

Basic View: State Window

The state window (cf. figure 5.12) will inform the user about states of processes if these processes try to access their critical section. This window should describe the affect of the double doorway mechanism.

The window shows four different areas represented by a rectangle and coloured according to the state of a process: light shaded green for state *wait1*, medium shaded green for state *wait2*, dark shaded green for state *collect* and red for *eat*. The rectangles are stapled one upon the other beginning with the area representing *wait1* and ending with the topmost area representing state *eat*. If a process gets interested in its resources

it will appear inside the area representing state *wait1* in form of a yellow coloured circle labeled with the process number. As long as a process remains in a certain state it will keep cycling around the respective area. When a process changes its state the process will move up to the next area. Finally, if a process finished with eating it will vanish.

The user can observe which processes are interested in its resources. Further it can be seen how long processes remain in certain states and how the double doorway mechanism really works. Also using the main view the user can conclude which processes prevent neighbour processes to proceed to the next state.

Communication View

According to previous resource allocation algorithms the communication view counts the number of request tokens which a process has needed in order to gain access to its critical section. When process p becomes hungry it will initialize the number of sent messages with zero. For each message m_1, m_2, m_3 (*request* messages will also be considered in case of the 4-fault-tolerant algorithm) the bar showing the number of sent messages increases by one unit. A mark will indicate the maximum number of those messages needed to enter the critical section among all attempts. The average bar shows the average number of sent messages for accessing once the critical section. A mark will also show the highest average value. All bars are coloured according to the colours associated with each process. Below every bar also the message size is displayed although it will remain constant during the whole execution of the algorithms.

When showing the animation of the δ-fault-tolerant algorithm the user will observe that the number of sent messages is proportional to the degree of the process as shown in the analysis. A process will send each time the same amount of messages in order to gain access to its critical section. This view was initialized with a maximum of 2δ messages.

Due to the different collection scheme of the 4-fault-tolerant algorithm the animation of this algorithm will show for each access to the critical section that numbers of needed messages may differ. The user will observe that more messages are needed than in the case of the δ-fault-tolerant algorithm, but increasing the degree of the communication graph will not lead to an exponential increase in number of messages since the analysis gave an upper bound of $O(\delta^2)$. The view was initialized with a maximum of $2\delta + \frac{1}{4}\delta^2$ messages. Although this value is not the upper bound it will give a good initial scaling and fit for most executions. The user is able to zoom in and out of the animation and thus can handle executions where more messages are needed.

Causality View

The causality view shows causal relations given by send and receive events considering the local time when events occurred. An arrow will point from sender to receiver starting at the local time of the sender when it sent the message and ending at the local time of the receiver when it received the message. The local clock time of a process will be changed when either a process changes its state or a message is received. When changing the state e.g. from *wait1* to *wait2* a process sets the local clock to

$$\textit{new value} := \textit{old value} + 1.$$

If a message is received the local clock will be set to

$$\textit{new value} := \max(\textit{old value}, \textit{received value}) + 1.$$

The colouring of a relation depends on the event that takes place. Messages m_1, m_2, m_3 are shown with the colour associated with the sender while for *request* message and a *fork* messages the colour associated with the receiver is used.

Process Step View

The process step view shows information about latest state, event, time and local clock for a process. The user can select a process by clicking inside the basic view the accordant node and retrace events for this process by clicking inside this view on buttons "Previous Event" or "Next Event". The functionality (see details on page 8) does not differ from other animations.

Process Occupation View

The process occupation view shows in real time or the time given by the simulator for each process how much time it spent in one of the states *wait1*, *wait2*, *collect* or *eat*. A bar for each process will be displayed at the time when a process is hungry or eats. For each state a different colour is used similar to the processes state in the basic view:

- A light shaded green will be used for the time period a process remained in state *wait1*.
- A medium shaded green will be used for the time period a process remained in state *wait2*.
- The bar is coloured in a dark shaded green for the period it remained in state *collect*.
- Red will be used when a process eats.

In order to describe which bar belongs to which process the user can observe marks at the beginning and at the end of each bar which are coloured according to the colour associated with each process.

Chapter 6

Counting Networks

For solving many multi processor synchronization problems it is important to have a good solution to the *counting problem*. It may even be the case that the synchronization problem itself can be expressed as a counting problem. The counting problem is to associate to a set of n tokens consecutive numbers by using a kind of fetch-and-increment mechanism. Many solutions to this problem perform poorly because of synchronization bottlenecks.

This chapter describes an efficient solution called *counting networks* (first introduced by Aspnes, Herlihy and Shavit [1]). Besides analyzing its properties, it gives a set of applications to well known multi processor synchronization problems; in particular, it shows how a sharable counter, a producer-consumer buffer and a barrier synchronization can be implemented by using counting networks. For one sort of counting networks called *periodic counting network* an animation will be introduced.

6.1 Properties of Counting Networks

Counting networks belong to a larger class called *balancing networks*. They consist out of simple two-input/ two-output elements called *balancer* (similar to the way that comparison networks consist out of two-input/ two-output elements called comparators). Basically, a balancer accepts tokens from its input wires and alternately outputs them to its upper and lower output wires.

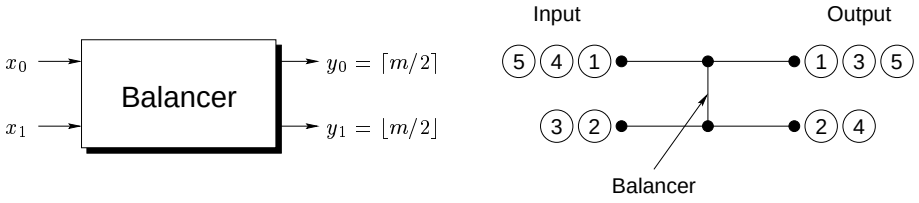

Figure 6.1: This figure shows the relation between the number of input tokens and the number of output tokens for a balancer. In the example of the right side one can see the output of tokens which were input in the sequence of their numbering.

In the following $x_i, i \in \{0, 1\}$ denotes the number of tokens having arrived at the

balancer's ith input channel, while $y_i, i \in \{0, 1\}$ denotes the number of tokens having arrived at the balancer's ith output channel (see also figure 6.1). The balancer is defined to be in a *quiescent state* when $x_0 + x_1 = y_0 + y_1$ holds. Further, it must satisfy the following properties:

1. In any state $x_0 + x_1 \geq y_0 + y_1$ holds.
2. Given any finite number of input tokens $m = x_0 + x_1$ to the balancer, it is guaranteed that it will reach in finite amount of time a quiescent state.
3. In any quiescent state $y_0 = \lceil m/2 \rceil$ and $y_1 = \lfloor m/2 \rfloor$ must hold.

Balancing Networks

A *balancing network* of width ω (cf. figure 6.2) is defined as a collection of balancers where output wires of balancers are connected to input wires of balancers such that no cycles occur. Tokens are input from ω designated input wires and output to ω designated output wires. Let x_i be the number of tokens entering the network at the ith input wire and y_i the number of tokens leaving at the ith output wire then for a balancing network the following safety properties must hold:

1. $\sum_{i=0}^{\omega-1} x_i \geq \sum_{i=0}^{\omega-1} y_i$
2. for any finite sequence of m input tokens, within finite time it will reach a quiescent state, i.e. one in which $\sum_{i=0}^{\omega-1} y_i = m$

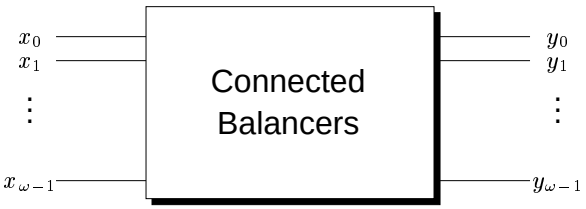

Figure 6.2: A balancing network of width ω. Hereby, x_i denotes the the number of input tokens on wire i, while y_i denotes the number of output tokens on wire i.

Timing Assumptions

For balancing networks there are not any timing assumptions made. The passing of tokens along the network is assumed to be completely asynchronous. In order to analyze the time a token needs from input to output of the network the *depth* of the network is an important parameter. The depth of the network is the depth longest directed path from an input wire to an output wire of the network. Under the assumption that a transition from input to output of a balancer takes at most time Δ, then any token will exit the network within time at most Δ times the depth of the network.

Counting Networks

Counting networks are balancing networks with a special property called *step property:* In any quiescent state of the network

$$0 \leq y_i - y_j \leq 1, \text{ for any } i < j,$$

must hold. The step property is another way to express that the network counts.

In the following some characteristics of counting networks will be introduced. The first characteristic given by lemma 6.1 shows better why counting networks are usable for counting. It is directly derived from the step property (see also figure 6.3). Lemma 6.2 expresses that every gap in the output sequence corresponds to some tokens which are still in the network, while lemma 6.3 shows a relation between counting networks and sorting networks. From this relation a lower bound for the depth of a counting network can be concluded (see lemma 6.4) since comparison networks that sort have smaller or equal depth than counting networks.

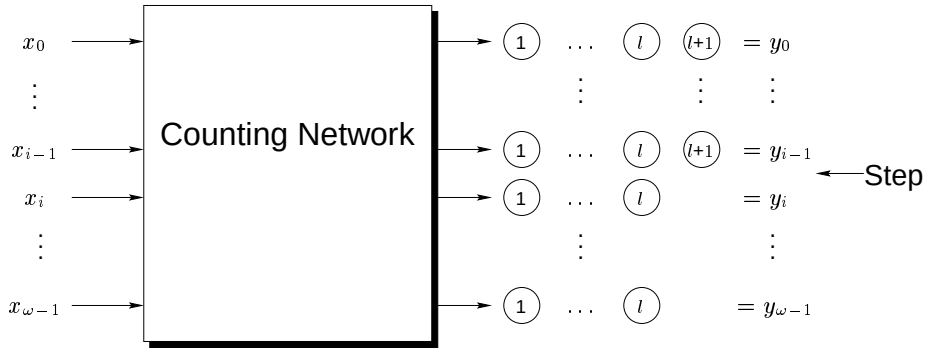

Figure 6.3: This figure should illuminate the step property for a counting network in a quiescent state. One can see the step in the number of output tokens between the $i-1$th and ith output where $0 \leq i < \omega$. For $i = 0$ there will not occur any step.

Lemma 6.1 *If* $\sum_{i=0}^{\omega-1} x_i =: m = \sum_{i=0}^{\omega-1} y_i$ *holds then* $y_i = \lceil \frac{m-i}{\omega} \rceil$.

Lemma 6.2 *Suppose that in a given execution a counting network with output sequence* $y_0, \ldots, y_{\omega-1}$ *is in a state where* m *tokens have entered and* m' *have left it. Then there exist non-negative integers* d_i*, such that*

$$\sum_{i=0}^{\omega-1} d_i = m - m'$$

and

$$y_i + d_i = \lceil \frac{m-i}{\omega} \rceil.$$

Proof. Let e be an execution where there do not exist such integers d_i. Then e can be extended to an execution e'that does not allow further tokens to enter the network. At the end of e' the counting network must reach a quiescent state in which the step

property does not hold. This is a contradiction to the definition of counting networks. □

Lemma 6.3 *If a balancing network counts, then its isomorphic comparison network sorts, but not vice versa.*

Proof.
"⇐": The Batcher's odd-even merging network is an example (cf. figure 6.4 for a sorting network which does not count.

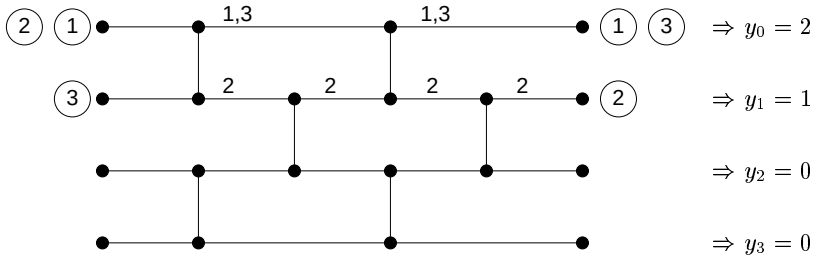

Figure 6.4: One can observe that the shown Batcher's odd-even merging network is in a quiescent state, but $y_0 - y_2 > 1$.

"⇒": In order to show that a balancing network sorts, the balancing network will simulate a comparison network. The network sorts if and only if it sorts all sequences of 0's and 1's. For this purpose, the following coding is applied to tokens:

$$0 \simeq \text{ a token} \qquad\qquad 1 \simeq \text{ no token}$$

The balancing network will be run in lockstep to the comparison network. By induction on the depth of the network it will be shown that the balancing network behaves in the same way as the comparison network.

$k = 0$: Obviously the input sequences are not changed.

$k \to k + 1$: Assume that the balancing network at depth k has the same behaviour as its isomorphic comparison network. Then at each balancer of depth $k + 1$ there will be a combination of $\{0, 1\} \times \{0, 1\}$ at its inputs. This is equivalent to one or none token for each input of a balancer. From comparing the four possible cases in figure 6.5 it results that the balancing network has the same behaviour than its isomorphic comparison network.

The step property of the balancing network assures that in the output there does not exist any gap of tokens (see lemma 6.1) and tokens representing 0's appear at the topmost outputs. Let k be the number of 0's, then for all $i < k$ $y = 1$ and for $i \geq k$ $y_i = 0$ hold. Hence, the network is sorting. □

Lemma 6.4 *The depth of any counting network is at least* $\Omega(\log \omega)$.

Proof. A comparison network needs at least depth $\Omega(\log \omega)$ to sort. According to lemma 6.3 also a balancing network will need at least depth $\Omega(\log \omega)$ to count. □

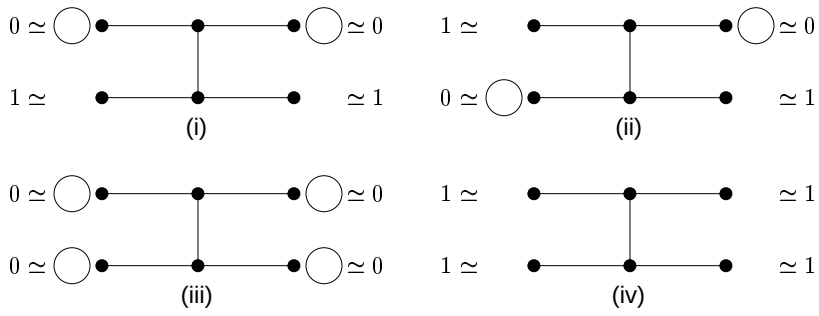

Figure 6.5: The four possible cases of inputs which can appear to a balancer of depth $k+1$ when simulating a sorting network.

6.2 The Bitonic Counting Network

In this section a specific counting network called *bitonic counting network* of depth $O(\log^2 \omega)$ is introduced. Hereby, the width of the network ω is always a power of 2. The network is recursively constructed from two bitonic networks of width $\frac{\omega}{2}$ which are merged together by an element called *merger* (cf. figure 6.6).

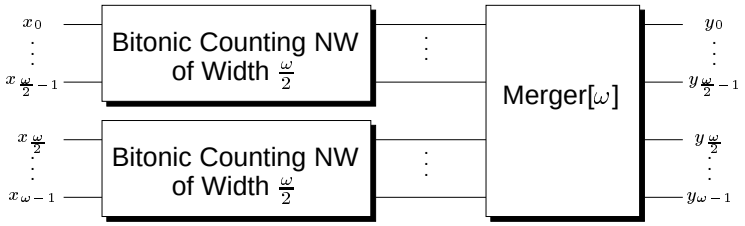

Figure 6.6: Construction of a bitonic counting network of width ω.

A merger$[2k]$ (cf. figure 6.7) has the property that it combines two input sequences $x := x_0, \ldots, x_{k-1}$ and $x' := x'_0, \ldots, x'_{k-1}$ (x_i and x'_i denote a number of tokens at position i) which both satisfy the step property and creates an output sequence $y := y_0, \ldots, y_{2k-1}$ which satisfies the step property in a quiescent state as well. A merger$[2]$ will be a simple balancer. For $k > 1$ a merger$[2k]$ is recursively constructed from two merger$[k]$ elements M and M'. It derives from M and M' output sequences $z := z_0, \ldots z_{k-1}$ and $z' := z'_0, \ldots, z'_{k-1}$ by applying to

- M the even subsequence $x_0, x_2, \ldots, x_{k-2}$ and the odd subsequence $x'_1, x'_3, \ldots, x'_{k-1}$.
- M' the odd subsequence $x_1, x_3, \ldots, x_{k-1}$ and the even subsequence $x'_0, x'_2, \ldots, x'_{k-2}$.

The output sequences z and z' are combined by sending z_i, z'_i through a final balancer which is connected to the merger's outputs y_{2i}, y_{2i+1}.

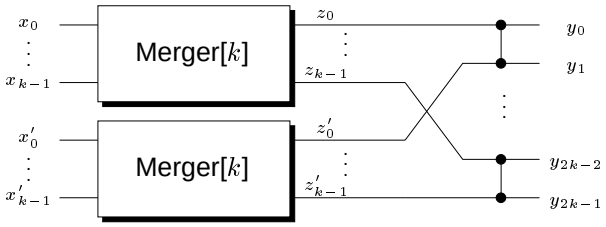

Figure 6.7: Construction of a merger$[2k]$.

Correctness

The construction of the bitonic counting network is correct if merger$[\omega]$ merges two sequences x and x' derived from the output of two bitonic counting networks of width $\frac{\omega}{2}$ such that in a quiescent state its output sequence y satisfies the step property.

Lemma 6.5 *If two input sequences x, x' of length k that satisfy the step property are input to the constructed merger$[2k]$ the output sequence y will satisfy the step property as well.*

Proof. By induction on k:

$k = 2$*:* The output sequence of a balancer is defined to satisfy the step property in a quiescent state.

$k > 2$*:* By the induction step the output sequences of the two merger$[k]$ elements M, M' will satisfy the step property if its input sequences satisfy the step property. Sequences x and x' satisfy the step property as their even and odd subsequences do. Hence, the output sequences z of M and z' of M' also satisfy the step property in a quiescent state. From

$$\sum_{i=0}^{k-1} z_i = \sum_{i=0}^{\frac{k}{2}-1} x_{2i} + \sum_{i=0}^{\frac{k}{2}-1} x'_{2i+1} = \left\lceil \sum_{i=0}^{k} \frac{1}{2} x_i \right\rceil + \left\lfloor \sum_{i=0}^{k} \frac{1}{2} x'_i \right\rfloor$$

and

$$\sum_{i=0}^{k-1} z'_i = \sum_{i=0}^{\frac{k}{2}-1} x_{2i+1} + \sum_{i=0}^{\frac{k}{2}-1} x'_{2i} = \left\lfloor \sum_{i=0}^{k} \frac{1}{2} x_i \right\rfloor + \left\lceil \sum_{i=0}^{k} \frac{1}{2} x'_i \right\rceil$$

one can derive

$$\left| \sum_{i=0}^{k-1} z_i - \sum_{i=0}^{k-1} z'_i \right| \leq 1.$$

Thus one can conclude that there exist at most one l for which

$$\max(z_l, z'_l) = \min(z_l, z'_l) + 1$$

and for all other $i \neq l$

$$z_i = z'_i$$

holds. Hence, in a quiescent state the last layer of balancers merging z_i, z'_i to outputs y_{2i}, y_{2i+1} guarantees the step property for output sequence y. □

Depth of the Network

The depth of a bitonic counting network of width ω is given by

$$\text{depth}(\text{bitonic counting network of width } \tfrac{\omega}{2}) + \text{depth}(\text{merger}[\omega])$$

From the recursive construction of a merger it follows that a merger$[\omega]$ has depth $\log \omega$. Hence, the depth of the bitonic counting network is due to its recursive construction $O(\log^2 \omega)$.

6.3 The Periodic Counting Network

A *periodic counting network* of width ω is built by concatenating $\log \omega$ subnetworks called *block*$[\omega]$ networks (cf. figure 6.8). Hereby, ω is a value which is chosen as a power of 2. The depth of $O(\log^2 \omega)$ for the periodic counting network is as good as the result for the depth of the bitonic counting network, but the construction is simpler since the whole network is constructed by only one element (the block$[\omega]$ network). Later it will be shown that this subnetwork can also be used for barrier synchronization.

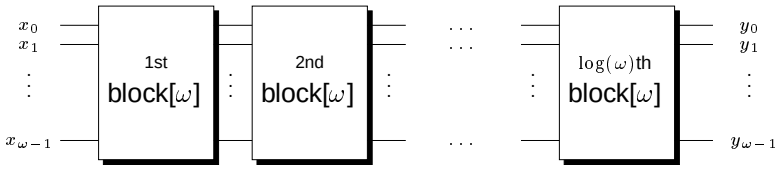

Figure 6.8: Construction of a periodic counting network of width ω.

The block$[\omega]$ networks are constructed by recursion (see also figure 6.9). A block$[2]$ is defined to be a balancer while a block$[2k]$ is constructed from two block$[k]$ networks M^A, M^B analogous to the mergers of a bitonic counting network. The input sequence of tokens

$$x := x_0, \ldots, x_{2k-1}, \text{ with } x_i \text{ is the number of tokens at input } i$$

is split in two subsequences x^A and x^B such that

$$x^A := \{x_i | \text{ index } i\text{'s two low order bits are } 00 \vee 11\}$$

and

$$x^B := \{x_i | \text{ index } i\text{'s two low order bits are } 01 \vee 10\}.$$

Sequence x^A is applied to M^A resulting in output sequence $z^A := z_0^A, \ldots, z_{k-1}^A$, while x^B is applied to M^B resulting in output sequence $z^B := z_0^B, \ldots, z_{k-1}^B$. Finally, z^A and z^B are combined by sending z_i^A and z_i^B to the input of a balancer which is connected to outputs y_{2i} and y_{2i+1}.

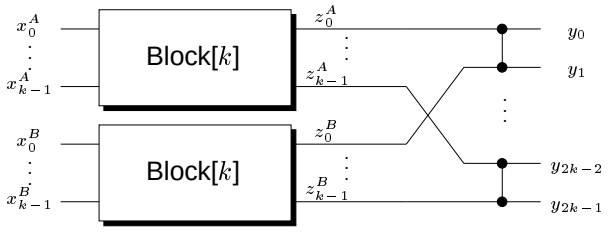

Figure 6.9: Construction of a block$[2k]$ network.

Correctness

Let a level l chain of a sequence $x := x_0, \ldots, x_{k-1}$ be defined as the subsequence

$$x^* := \left\{ \quad x_i \quad \middle| \quad \begin{array}{l} x_i \in x \text{ such that the } l \text{ low order bits are identical for} \\ \text{all other index } j \text{ with } x_j \in x^* \end{array} \right\}$$

The basic idea of this construction is that in a quiescent state all level $l-1$ chains of the output of a block$[k]$ will satisfy the step property if all level l chains of the input sequence satisfy the step property. For example, the even subsequence x^E and the odd subsequence x^O are level 1 subsequences of x. If x^E and x^O satisfy both the step property and x is input to a block$[k]$ network then the output, a level 0 chain, must satisfy the step property in a quiescent state as well. Initially, in a periodic counting network each input x_i is a sequence which satisfies the step property. Hence, in a quiescent state all level $\log\omega$ chains satisfy the step property before the first block$[\omega]$, all level $\log\omega - 1$ chains satisfy the step property before the second block$[\omega]$ and finally all level 1 chains satisfy the step property before the $\log\omega$th block$[\omega]$ which explains why $\log\omega$ block$[\omega]$ networks are used. The correctness follows now directly from lemma 6.6.

Lemma 6.6 *Let block$[k]$ be in a quiescent state with input sequence x and output sequence y. If all level l input chains to a block have the step property, then this is the case for all $l-1$ output chains.*

Aspnes, Herlihy and Shavit prove this by induction starting with the case $i = 1$ which is the above example of even and odd subsequences. The proof technique is similar to the correctness proof of the bitonic counting network. The interested reader is referred to the paper of Aspnes, Herlihy and Shavit [1].

Depth of the Network

Due to its recursive construction a block$[\omega]$ has depth $O(\log\omega)$. Taking $\log\omega$ block$[\omega]$ networks one after the other leads to depth $O(\log^2\omega)$.

6.4 Applications of Counting Networks

The following part deals with practical applications for counting networks. In most systems solutions are implemented by using lock mechanisms. From practical experi-

ments counting networks seem to outperform these methods (see Aspnes, Herlihy and Shavit [1]).

Previous chapters dealt only with algorithm that allowed message passing. For simplicity in the following description it is assumed that processes may access shared memory locations. However, a periodic counting network was simulated and animated according to the message passing model where each balancer is a process with two input and two output links.

6.4.1 Shared Counter

A *shared counter* is an element which can be used to apply to a set of tokens a consecutive numbering. A shared counter can be realized by a counting network of width ω where each output i is associated with an integer cell c_i and $c_i := i$ initially. A token enters the network at an arbitrary input wire and will leave the network at some wire j. Then the token sets in one atomic step its number to c_j and increases $c_j := c_j + \omega$.

Correctness

If a finite set of tokens is applied to the counting network, the network is guaranteed to reach a quiescent state. Hence, the number of output tokens will satisfy the step property which assures that tokens have a consecutive numbering.

Time Complexity

Let Δ denote the longest time for a token to traverse any balancer, and δ denote the depth of the network, then a token needs $O(\delta\Delta)$.

6.4.2 Producer/Consumer Buffer

The producer/consumer problem can be described by m producer processes which insert an item into a buffer and m' consumer processes which remove an item from this buffer. A solution to this problem uses two counting networks of width ω and a ω-element buffer *buff*$[0 \ldots \omega - 1]$. The buffer cell buff$[i]$ is associated with the ith output wire of both counting networks.

The first counting networks is used by producers. A producer enters this network at an arbitrary input and will exit the network at some output wire j. If buff$[j]$ is empty the producer will insert its item. Otherwise it waits with inserting its item until buff$[j]$ is empty.

The second counting network is used by consumers. A consumer enters this network at an arbitrary output and will exit the network at some wire j. If there exist an item in buff$[j]$ it will consume the item. Otherwise, it will wait until there is an item to consume.

Assume that the time to update a buff$[j]$ is negligible. If $m \leq m'$ then the step property will guarantee that each produced item will find a consumer. Every producer will leave the network in time $O(\delta\Delta)$. Analogous for $m \geq m'$ the step property

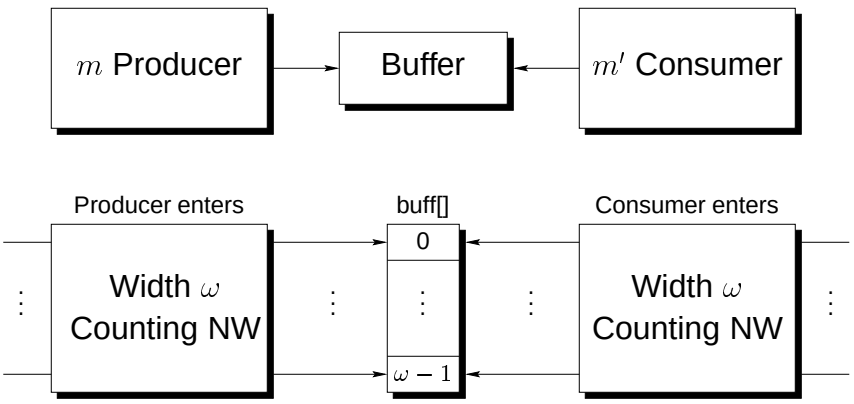

Figure 6.10: Producer/consumer Buffer with counting networks

guarantees that all consumers will find an item to consume. Hence, every consumer will leave the network in time $O(\delta\Delta)$.

6.4.3 Barrier Synchronization

A barrier is a data structure which ensures that no process advances beyond a particular point in a computation until all processes have arrived at that point.

The properties of a counting network imply an easy implementation. After a process finished a step in computation it sends a token through a counting network. If one token leaves the network with value $n-1$ it can be concluded that all processes sent a token through the network and consequently finished its computation step.

However, barrier synchronization can be achieved by subnetworks of counting networks which are called *threshold networks*. A threshold networks of width ω is a balancing network with input sequence $x_0, \ldots, x_{\omega-1}$ and output sequence $y_0, \ldots, y_{\omega-1}$ such that in any quiescent state

$$y_{\omega-1} = m \Leftrightarrow m\omega \leq \sum x_i < (m+1)\omega$$

holds. The counting network itself is a threshold network, but also the merger$[\omega]$ of the bitonic counting network and the block$[\omega]$ of the periodic counting network are threshold networks. They detect each time when ω tokens traversed through the network.

Constructing a Barrier with a Threshold Network

Assume there are processes $p_0, \ldots, p_{n-1}$ which are synchronized by barriers. Then, the barriers separate the algorithm in phases $\pi_0, \pi_1, \pi_2, \ldots$. A barrier is implemented by a threshold network of width ω such that

$$n \equiv 0 \bmod \omega.$$

Each output is associated with shared variable c_i, $c_i := i$ initially. Further all processes can access a shared variable F which is initialized with value *FALSE*. Each process

knows the local variable *sense* such that

$$\textit{sense} \quad = \quad \textit{TRUE} \text{ in even phases } \pi_0, \pi_2, \pi_4, \ldots$$

and

$$\textit{sense} \quad = \quad \textit{FALSE} \text{ in odd phases } \pi_1, \pi_3, \pi_5, \ldots$$

The processes execute the algorithm in the following way:

1. If a process p_i finished with its computation for a phase it will enter the network at some arbitrary input wire and finally exit at output wire j.
2. Process p_i sets $v(p_i) := c_j$ and $c_j := c_j + \omega$ in one atomic step.
3. Case $v(p_i) = (n-1) \bmod n$: Process p_i left the network at wire $\omega - 1$ since $\omega | n$. This implies that all other processes have already entered the network and are finished with its computation for this phase. Hence, p_i sets $F := \neg F$ and *sense* $:= \neg$*sense* before it continues with the next phase.
4. Case $v(p_i) \neq (n-1) \bmod n$: Process p_i does not know whether all other processes entered the network. Hence it waits until *sense* $= F$ holds before it sets *sense* $:= \neg$*sense* and continues with next phase.

6.5 Animation of the Periodic Counting Network

The animation of the periodic counting network is built according to the general considerations in chapter 2. The user can choose among different widths for the counting network. Further one can choose between an *interactive* and a trace file animation. In the interactive animation the user selects with the mouse either to produce a token or force a balancer to send one of its token waiting to proceed to an output. The trace file animation built animation frames according events of a trace file as this was the case for previously discussed animation. Both animations have the same appearance so that the description concentrates on the trace file animation and points at differences. One difference to other animations is that the basic view is split in three windows. The *main window* shows the counting network and tokens which move between balancers. The *input window* informs about the number of tokens produced for each input wire, while the *output window* informs about which tokens vanished at which output.

Basic View: Main Window

The main window shows processes, each representing a balancer, as arrows connecting the wires from which they receive tokens and to which they output tokens. The arrow points to the wire where the next token will be output. All balancers are placed on ω wires such that they satisfy the construction of the periodic counting network (see figure 6.11).

Tokens are shown as yellow circles with an identifier which depends on the order of appearance. They appear on one of the input nodes and travel from balancer to

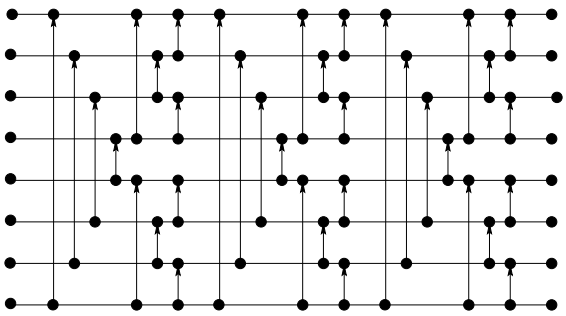

Figure 6.11: Placement of balancers for a periodic counting network of width 8.

balancer according to the algorithm's events until tokens reach an output node where they vanish.

In the interactive animation clicking on an input node of the network causes a token to appear at this input and to move to the input of the next balancer. If there are any tokens at a balancer clicking this balancer will cause a token either to move to the next balancer or to move to an output of the network.

Basic View: Input Window

The input window shows for each input of the network the number of tokens which have been produced so far. Further the input node of which the last token appeared is marked by an arrow.

Basic View: Output Window

The output window illuminates gaps in the output sequence of tokens. For each output, tokens with their identifier appear in the output window after vanishing in the main window. If all output nodes reached level k i.e all output nodes produced k tokens then the first k tokens which appeared inside the output view will disappear in a "tetris manner". The user should observe that in a quiescent state of the network the step property holds.

Communication View

The communication view counts for each balancer the number of sent tokens indicated by a bar chart. Each bar is coloured according the colour associated with each balancer. Moreover an additional bar indicates the average of sent tokens. Although message sizes are constant they are displayed below the accordant bar.

After sending tokens among the network the user can observe that the number of sent tokens for balancers which are closely to the output is almost the same then the average number of sent tokens. The maximum number of sent messages depends on the number of produced tokens. Hence, it is not possible to specify an upper bound for the expected number of messages before the end of the animation. For this reason a user might have manually to resize the window size.

Causality View

The causality view shows the causal relation of the algorithm induced by a balancer sending tokens and receiving tokens. For each relation an arrow appears beginning at local time when a token is sent and ending at local time when a token is received. The local clock is increased on receiving a token such that its new value is set to

$$\max(\textit{send\_tm}, \textit{old value of local clock}) + 1.$$

Hereby *send_tm* denotes the sender's local clock when sending the token. The relations are coloured according to the colour associated with the sender of the token.

Process Step View

The process step view shows for a balancer latest information about events. A balancer can be selected by clicking inside the main window of the basic view on a balancer. The sequence of events can also be retraced by clicking on buttons "Previous Event" or 'Next Event" of this view. The functionality (see details on page 8) does not differ from other animations.

Process Occupation View

The process occupation view shows in real time how long a balancer was busy i.e. the time tokens were waiting at a balancer to proceed. When a token arrives at a balancer where no other tokens wait to proceed then a bar for this balancer will be created starting at arrival time of the token. The bar ends when the balancer enters the state again where no tokens wait to proceed. Every bar is coloured according to the colour associated with each balancer.

Chapter 7

Implementation

7.1 Introduction

The practical part of this master thesis consists out of building animations for the algorithms described in previous chapters, the implementation of these algorithms for the simulation environment of LYDIAN and the implementation of an interface which enables users to create network description files for simulation. The code was implemented in C++ and C. For building the animations a C++ library called POLKA was used. The algorithms were implemented by writing procedures in C and combining these procedures with transitions of events supported by the simulator. For the implementation of the interface creating network description files, graphwin of the LEDA library was used. The introductory part will explain the main concepts of these libraries. The understanding will be helpful for reading the elaboration of each implementation aspect introduced in the next sections.

7.1.1 POLKA: A Library for Building Animations

POLKA [14] is a library of visualization and animation routines, useful to visualize sequential and concurrent algorithms. It offers a class called *animator* which controls a set of animation windows, called *view*, and *animation objects* inside a view. An animation object can be any kind of graphical object e.g. a circle, a rectangle, a text etc. Views and animation objects are organized as classes as well.

The animator controls the flow of significant events of an algorithm. According to each event the programmer can define *actions* which modify an animation object defined inside a view. For each action the animator will compute for the set of views a series of animation frames such that for every time unit there exist for each view a frame. The animator can be requested to show for a period of time the accordant frames. The frames will be shown one after the other such that all views evolves simultaneously. Since the creation of frames allows to apply more than one actions to animation objects inside a view, an observer will receive the impression of parallelism. For example two objects could be moved at the same time.

7.1.2 DIAS: The Simulator of LYDIAN

In chapter 1.3 the environment LYDIAN was already introduced. This subsection deals only with the creation of programs for the simulator of LYDIAN called DIAS.

During the execution of a program the simulator chooses pending events of a process and schedules them. The execution of these events will trigger the execution of procedures that will use the type of the event and the local state of the process as an input and will produce the new state of the process. Therefore, a programmer must define procedures and transitions such that for a state and an event the accordant procedure will be called. Hence the simulator also needs information about possible states and messages which are transmitted among the system.

The simulator knows following events for a process p:

- *RECMES(TYPE):* p receives a message of type TYPE. The simulator will register this event if some process sends a message during its local computation.
- *INITPROTOCOL:* p will be waken up. The user can define by creating the network description file which processes of the communication graph should be waken up when starting the protocol.
- *TIMEOUT_X:* p started in some previous local computation a timer and timeout activates p again. The simulator offers 5 different timers.

With information about states, messages, procedures and transitions the LYDIAN environment offers the possibility to create an executable program. This program will be able to run with different network description files i.e. different communication graphs and different timing assumptions. The output of the program can be used for visualization.

7.1.3 LEDA: Library of Enhanced Data Structures and Algorithms

LEDA [10] is a library which supports many data structures and algorithms. For the creation of network description files a tool was needed which supports an easy way of drawing graphs and manipulating the structure of graphs. With every vertex of the graph representing a process timing assumptions must be associated. Further with every edge representing a link between two processes message transmission times must be associated. The graphwin utility of LEDA supports these properties. It offers algorithms for manipulating graph structures and allows programmers to manipulate the graphical user interface.

7.2 Animation Using POLKA

All animations work according to the same principle. The main program offers a graphical interface for interaction with the user and reads events from a debug file created by some simulation. The events are sent to a control unit, called *controller*. The controller will cause the animator to create for all animation windows (views)

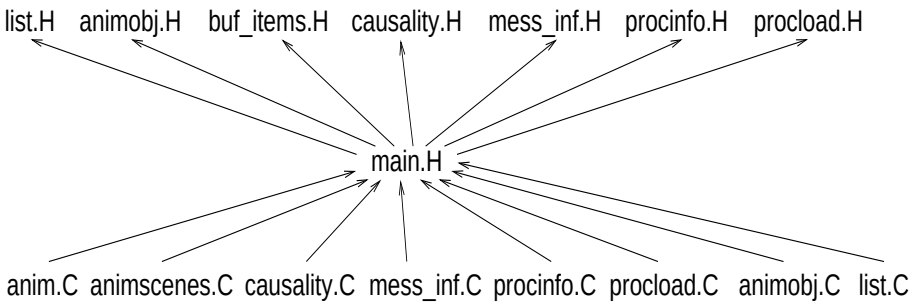

Figure 7.1: This figure shows the used program modules and their dependencies for building an animation.

frames and ensures the frames to be displayed smoothly one after the other. The user selects via the graphical interface the speed of which the animation evolves.

Due to the structure of animations introduced in chapter 2 some views will have the same functionality for different animations. Therefore, the views are structured in independent program modules which can be integrated in any animation. This allows adding further modules and easier development of new animations although the basic view may differ. Figure 7.1 shows the dependencies of the used program modules which are introduced in the next paragraph.

The program modules

anim.C: ”anim” denotes the name of the animation. This module contains the main program which initializes the graphical interface and registers all views. After initialization it reads the input events which will be transmitted to the controller of the animator.

animscenes.C: As before ”anim” denotes the name of the animation. The module includes all functions provided by the class animator. Further it includes all functions of the class realizing the basic view. The animator functions evaluate the events which have been transmitted by the main program. According to events it creates in cooperation with the registered views new animation frames and initiates showing frames for a period of time (depending on the event). Note that functions for animator and basic view will differ for each animation.

causality.C: This module includes all functions of the class which realizes the causality view. The call of the class’s public functions will modify and create frames according to the functions parameters. Finally, the animator can initiate the showing of frames if this view was registered.

mess_inf.C: The class realizing the communication view is included in this program module. It supports modification and creation of frames for this view.

procinfo.C: This module contains the public functions of the class realizing the process step view. Again the class supports modification and creation of frames for this view.

procload.C: It includes the functions of the class realizing the process occupation view supporting modification and creation of frames for this view.

animobj.C: It includes new defined animation objects which realize undefined animation objects of the POLKA library.

list.C: This module supports a dynamic linked list which is used in almost all other program modules.

From this general structure of program modules the following parts will move on to a more detailed description of classes and data structures realized in each module. Hereby, the beginning deals with those mechanisms which are common for all animations before describing specific elements for each animation.

7.2.1 Classes of the Animation

All views of the animation and the animator itself are implemented as separate classes which can be used for any animation. The implementation of classes uses given classes from the POLKA library and derives subclasses.

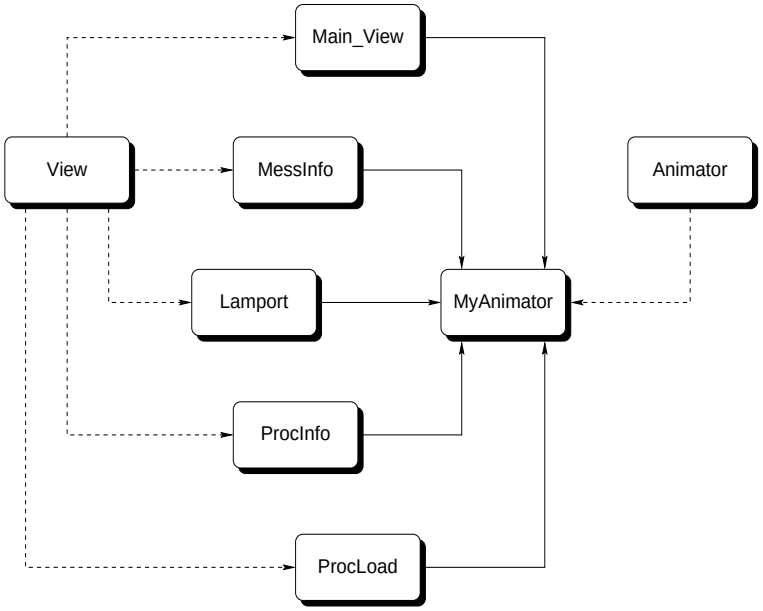

Figure 7.2: This figure shows the dependencies between used classes. A solid arrow pointing from class X to class Y means that an instance of class X is used inside class Y. A dashed arrow arrow from class X to class Y shows that class Y is a subclass derived by class X.

The dependencies of the used classes are shown in figure 7.2. The following part of this section introduces these dependencies.

7.2.1.a Animator

The animator is the control center of each animation. It controls the showing of frames inside its registered views. Further it creates or modifies animation frames according to events transmitted by the main program. The POLKA environment provides a class called Animator for registering names and types of algorithm events, sending algorithm events to a control unit and animating frames of registered views. From this class a subclass called MyAnimator was derived in which the control unit, functions for computation of animation frames and views were defined. MyAnimator is structured in the following way:

```
class MyAnimator : public Animator
{
private:
   Main_View net;                                   /* basic view */
   Lamport causal;                                  /* causality view */
   ProcInfo procinf;                                /* process step view */
   MessInfo messinf;                                /* communication view */
   ProcLoad procload;                               /* process occupation view */

   /* Initialization of views */
   int Init_Views();

   /* functions according algorithm events */
   ...

   /* functions which support animations using a communication graph */
   ...

public:
   MyAnimator();
   int Init_Graph();
   int Controller();
   int Time();
};
```

views:

The animator object knows for each view described in chapter 2 an object (net, causal, procinf, messinf and procload) for the modification of each view. The respective classes will be introduced in the following subsections.

int `MyAnimator::Init_Views()`

This routine will cause each view to be initialized. Each view will show its initial objects.

event functions:

These functions have to be designed according to each animation. They will change the animation objects inside each view according to the events of the algorithm. This is realized by calling the provided public functions of each view object. The only exception is given by the basic view object called *net* of which the animation objects are controlled directly by the animator.

graph functions:

Some animations will have to consider the structure of the communication graph. In this case the animator object will support the respective functions.

`MyAnimator::MyAnimator()`

This constructor of the subclass *MyAnimator* registers the previous defined view objects and causes the basic view to appear on the screen. For all other objects references of each view are assigned to global variables in order to control the visibility of these views by a graphical interface. Finally, the private variables are initialized.

int `MyAnimator::Init_Graph()`

This function is used to read the structure of a communication graph. Naturally this function is only used if the animation deals with such a graph.

int `MyAnimator::Controller()`

The *controller* of the animator decides which event functions are called when an algorithmic event happened. Further it causes all not shown frames which happened before the algorithmic event to be shown.

int `MyAnimator::Time()`

The function informs about the time when the latest algorithmic event happened. This value will be changed by the animation when frames for an event are created.

7.2.1.b Basic View

The basic view is implemented as a subclass derived from the class *View* of the POLKA library. It provides the window in which the main animation is shown. It offers functions to open the window and change a time parameter indicating from which frame on modification or creation of frames is allowed. The class *Main_View* has the following structure:

```
class Main_View : public View
{
public:
   void Open_Window();
   int Time();
   int NewTime(int);
};
```

void `Main_View::Open_Window()`

The animation window will be generated and will appear on the screen.

int `Main_View::Time()`

The time of an animation frame is returned with the property that it is the first frame guaranteed not to be shown on screen when calling this function. This function can be used by the animator in order to select the frame to be shown next. Note that this function differs from the time-function of MyAnimator which informs about the time of the latest algorithm event.

int `Main_View::NewTime(`**int** `newtime)`

Parameter $newtime$ specifies the time that indicates the first frame which has not been animated yet. This function can be called by the animator after initiating frames to be shown.

All animation objects which will appear inside the basic view are controlled by the animator's event functions. For this reason this view will provide the same functions for all animations, but it will look quite different. The functions for information and modification the time of the basic view are important for selecting the frames to be shown by the animator.

7.2.1.c Communication View

Also the class *MessInfo* is a subclass derived from the class View provided by POLKA. In contrast to the class Main_View it manages all animation objects. For modification it offers public functions which allow the animator to change the communication view according to algorithm events. All animations use the following structure of the communication view:

```
class MessInfo:public View
{
public:
   MessInfo();
   void Init(int, int, int);
   void Increase(int, int, int);
   void Reset(int, int, int, int);
   void ChangeAv(int, int);
   void MessSize(int, int, int, int);
private:
   ...
};
```

`MessInfo::MessInfo()`

The function realizes the constructor of the class.

void `MessInfo::Init(`**int** `frametime, ` **int** `max_proc, ` **int** `max_mess)`

The initial layout for the communication view will be generated. This includes also the initialization of all variables and animation objects. Hereby frametime denotes the animation frame for which the layout will be created, max_proc the number of used processes and max_mess the maximum number of messages which will be sent. The values max_proc and max_mess will be used to create a layout which fits to the animation. However, it is possible to send more messages for a process than specified, but in this case the user might have to manually rearrange the shown animation window to see everything of the animation.

void `MessInfo::Increase(`**int** `frametime, ` **int** `id, ` **int** `send_tm)`

For the animation frame given by $send\_tm$ an increase by one unit in messages will be shown for the process identified by id. Also the average bar will be modified according to this event at $send\_tm$. The user will see the accordant rectangles being increased. The variable $frametime$ denotes the next possible frame which can be modified. It is used for deletion of old actions and animation objects which not needed for frames to be animated at time $\geq frametime$.

void `MessInfo::Reset(`**int** `frametime, ` **int** `id, ` **int** `send_tm,`
int `average)`

The sent messages of process identified by id will be set to 0 at $send\_tm$. With parameter $average$ it can be specified how the computation of the average of sent messages is computed. If this value equals the number of processes the average of the actual displayed messages is shown.

However, in many animations the resetting of messages means that the process starts with a new job. Then it is interesting to know the average number of messages among all processes necessary to finish a job. In order to achieve this, the average value should contain the actual number of jobs. Note that during an animation the mix of both methods would lead to wrong results.

The parameter $frametime$ denotes the next possible frame which can be animated. As in the previous function it is used for deletion of old animation objects and old actions.

void `MessInfo::ChangeAv(`**int** `average, ` **int** `frametime)`

This function allows to modify directly the way how the average is computed. Hereby the value $average$ has the same meaning as described in function *Reset*. Parameter $frametime$ specifies the animation frame for which the change will be valid.

void `MessInfo::MessSize(`**int** `frametime, ` **int** `id, ` **int** `size,`
int `send_tm)`

This function allows to specify the message size of the process identified by parameter id. The new message size given by parameter $size$ will be valid at animation frame specified by $send\_time$. According to previous functions parameter $frametime$ denotes the next possible animation frame which can be animated.

7.2.1.d Causality View

The causality view was taken from an tool called PVaniM and modified according to the requirements of the constructed animation. The class *Lamport* is a subclass derived from the class *View* provided by POLKA. All animation objects of this view are controlled by class *Lamport*. Public functions allow the animator to communicate with this view. Following structure was applied to *Lamport*:

```
class Lamport : public View
{
public:
   Lamport()
   int Init(int);
   int Send(int,int,int,int,int,int);
   int Receive(int,int,int,int,int,int);
private:
   ...
};
```

int `Lamport::Init(int c_nvprocs)`

The animation window including the initial layout will be created. Further the animation objects will be initialized.

int `Lamport::Send(int myprocid, int clockval, int polkatime,`
`int friendprocid, int type, int total)`

Inside the animation window a relation will be shown at time $polkatime$. The relation will be indicated in form of a small arrow which points from process identified by $myprocid$ into direction to the process identified by $friendprocid$, but not touching this process. Further a circle located at the position of the sender indicates the message which is sent along the link. The size of this circle is indicated by parameter $total$. Parameter $clockval$ denotes the local clock of the sender. With parameter $type$ the message type is defined. The type influences in which colour the arrow will be shown.

int `Lamport::Receive(int myprocid, int clockval,`
`int polkatime,int friendprocid,`
`int type, int total)`

Receiving a message defines the end of the relation. At frame given by $polkatime$ the top of the arrow which was created by the accordant send event will grow to the location defined by parameter $clockval$ and parameter $procid$. Hereby $clockval$ denotes the local clock's time when the message arrives at the receiver. The sender is specified by parameter $myprocid$, while the receiver is specified by parameter $friendprocid$. The message is identified by parameters $type$ influencing also the colour of the arrow and $total$ defining the message's size. The call of this function will animate the circle moving along the arrow to the receiver beginning at animation frame defined by $polkatime$.

7.2.1.e Process Step View

The process step view is implemented as a subclass derived from POLKA's class *View*. It offers public functions which allow modification of the view. The class *ProcInfo* has following structure:

```
class ProcInfo: public View
{
public:
   ProcInfo();
   void Init(int, int);
   void Show(void*, int, int);
   int IsShown(int);

   Button* prev;
   Button* next;

private:
   ...
};
```

`ProcInfo::ProcInfo()`

The constructor initializes variables and data structures.

void `ProcInfo::Init(`**int** `frametime,` **int** `processes)`

Inside the animation window the initial layout will be created at $frametime$. Parameter $processes$ denotes the number of used processes. Further with the animation objects $prev$ and $next$ a call-back function will be associated. Clicking with the mouse on one of these buttons will call the accordant functions and show for a selected process either the previous or next event.

void `ProcInfo::Show(`**void** `* data,` **int** `frametime,` **int** `send_tm)`

Parameter $data$ defines the event which will be shown at $send\_tm$. If the value of $data \geq 0$ then $data$ specifies the latest event of the process whose identifier equals the value of $data$. Otherwise the previous event (case $data = -2$) or the next event (case $data = -1$) of the already displayed event will be shown. Parameter $frametime$ denotes the next possible frame which can be animated.

int `ProcInfo::IsShown(`**int** `id)`

This function returns for a process identified by parameter id whether events of this process are shown (1 is returned) or not (0 is returned).

7.2.1.f Process Occupation View

The process occupation view is derived by a subclass of POLKA's class *View*. Public functions allow modification of the view. Following structure is applied to class

ProcLoad:

```
class ProcLoad:public View
{
public:
   ProcLoad();
   void Init(int, int, int);
   void Send(int, int, COLOR, int);
   int Receive(int, int, int, COLOR, int, int);
   void Create_Start(int, COLOR, int);
   void Create_Grey_Start(int, COLOR, double, int);
   int End(int, int, COLOR, int);
   int Grey_End(int, int, int);

private:
   ...
};
```

`ProcLoad::ProcLoad()`

The constructor of the class initializes private variables.

void `ProcLoad::Init(`**int** `frametime,` **int** `max_proc,`
int `expect_time)`

The initial layout will be created at $frametime$ for max_proc processes. The parameter $expect\_time$ denotes the period of time shown inside the window. If some bars cross this value the view will automatically scroll to the end of the bar.

void `ProcLoad::Create_Start(`**int** `id, COLOR col,` **int** `send_tm)`

For the process identified by parameter id a new bar will be created at $send\_tm$. The mark which colours the beginning of the bar will be coloured according to parameter col. Its beginning will be located at the position defined by $send\_tm$ and id.

void `ProcLoad::Create_Grey_Start(`**int** `id, COLOR col,`
double `intensity,`
int `send_tm)`

For the latest existing bar of the process identified by parameter id a new colour given by col will be associated at $send\_tm$. Parameter $intensity$ allows to modify the intensity of the given colour by applying values between 0 and 1.

void `ProcLoad::Send(`**int** `frametime,` **int** `id,`
`COLOR col,` **int** `send_tm)`

If there does not exist any bar for the process identified by parameter id a new bar coloured according to parameter col will be created at $send\_tm$. Otherwise the latest bar of this process will grow to the location defined by $send\_tm$ and id. This action will start at the next possible frame which can be animated (this is defined by $frametime$) and be finished at $send\_tm$.

int `ProcLoad::Receive(`**int** `frametime, int id, int from,`
`COLOR col, int send_tm, int rec_tm)`

This function has the same functionality than defined by `ProcLoad::Send`, but also supports showing the relation between sender and receiver. Additional an arrow will be shown pointing from the sender at $send\_tm$ to the receiver at rec_tm. Hereby the sender is identified by parameter id, while the receiver is identified by rec. Parameter $frametime$ denotes the next possible frame to be animated and is used for deletion of old objects and actions.

int `ProcLoad::End(`**int** `frametime, int id, COLOR col, int end_tm)`

The end of a bar for the process identified by parameter id will be marked at end_tm. Therefore the latest bar of this process will increase to the position given by end_tm and id. The action will start at the first possible frame to be animated and will end at end_tm.

int `ProcLoad::Grey_End(`**int** `frametime, int id, int end_tm)`

The latest bar of process identified by id will increase to the location defined by id and end_tm. The action will start at the first possible frame to be animated and end at end_tm.

7.2.2 The Graphical Interface

The graphical interface offers two windows. The *"Polka Control Panel"* is directly invoked by the animator and allows to tune the speed of the animation, execute the animation in a step by step mode or to halt an animation. Every event of this window directly effects the animation.

The *"Animation Control Window"* is implemented in *Motif* [6]. It offers the user the possibility to select which of the offered views appear on the screen. A view can be caused to appear or disappear at any time of the animation. The interface communicate with a view by accessing references of global variables which are initialized by the animator. The following functions realize the "Animation Control Window":

void `console()`

Creates the window and initializes with each button a call-back function.

void `toggled(Widget w, void *f1, void *f2)`

This function is called when a box of the window was toggled. Parameter $f1$ identifies the box and $f2$ returns the state of the box. According to parameters $f1$ and $f2$ the selected view will appear or disappear on the screen.

void `quitCB(Widget, XtPointer, XtPointer)`

The quit button in one of the graphical interfaces was selected. The program will quit the animation.

7.2.3 Main Program and Events

The main program creates an instance of *MyAnimator* called *anim* (for every animation this variable is named after the animation name) and initializes the graphical interface by calling the appropriate function. Afterwards all possible algorithmic events are registered for the animator. For this purpose the function called *RegisterAlgoEvt* provided by class *Animator* is used. Hence, the animator knows which events are expected and in which format they are transfered.

The main program filters algorithm events from the input and transfers them to the animator by the use of function *SendAlgoEvt* also provided by class *Animator*. The events will cause the animator to compute the respective animation frames. The program will terminate if the user clicks the *quit button* provided inside the graphical interface.

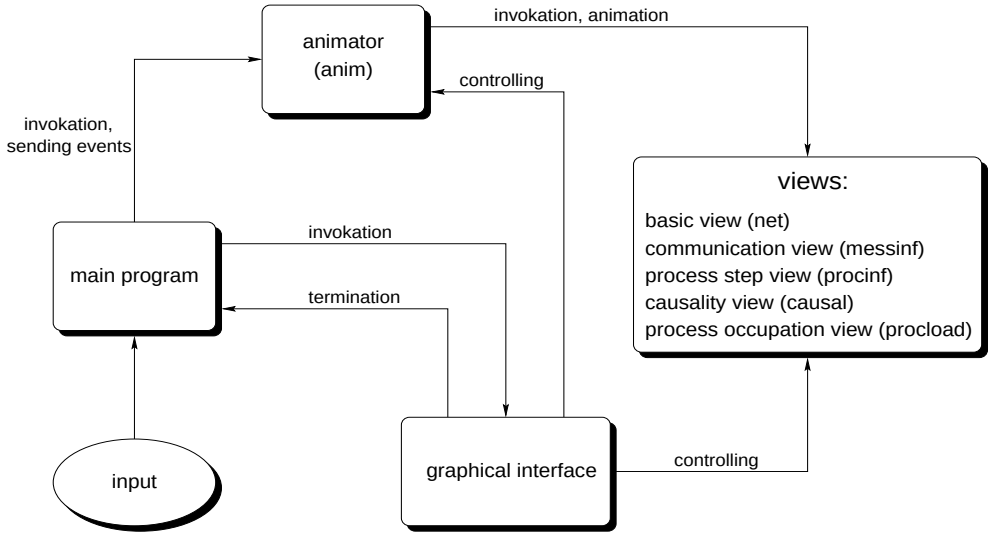

Figure 7.3: This figure shows the flow of information among the described components.

The flow of information during the execution of the program can be seen in figure 7.3. The algorithmic events of the animation differ a lot. In the following the algorithm events of each animation are introduced. Hereby the events are structured in the order: *input event* implies an *animator event* which causes an *event function* of the animator to be called.

7.2.3.a The Broadcast with Acknowledgement Algorithm

The animation of the broadcast with acknowledgement algorithm recognizes following events and adds to the class *MyAnimator* following event functions:

`START i j` $\rightarrow$ `Start i j`
$\rightarrow$ **int** `MyAnimator::Start(`**int** $id = i$, **int** $start\_tm = j$)

Process with identifier id initiates the broadcast with acknowledgement algorithm.

SEND_BROADCAST i j k → SendBrdcast i j k
→ **int** MyAnimator::Send_Brdcast(**int** $id = i$, **int** $adj = j$,
int $start\_tm = k$)

Process with identifier id sends a *broadcast* message to process identified by adj at $send\_tm$.

REC_BROADCAST i j k → RecBrdcast i j k
→ **int** MyAnimator::Rec_Brdcast(**int** $id = i$, **int** $from = j$,
int $start\_tm = k$)

Process with identifier id receives from process with identifier j a broadcasted message at $start\_tm$.

SEND_ACKNLDGEMNT i j k → SendAcknldgemnt i j k
→ **int** MyAnimator::Send_Acknldgement(**int** $id = i$, **int** $to = j$,
int $start\_tm = k$)

Process with identifier id acknowledges to process with identifier to that a broadcasted message was received. The acknowledgement is sent at $start\_tm$.

REC_ACKNLDGEMNT i j k → RecAcknldgemnt i j k
→ **int** MyAnimator::Rec_Acknldgement(**int** $id = i$, **int** $from = j$,
int $start\_tm = k$)

Process with identifier id receives an acknowledgement from process with identifier $from$ at $start\_tm$.

END_WORK i j → EndWork i j
→ **int** MyAnimator::End_Work(**int** $id = i$, **int** $start\_tm = j$)

Process with identifier id received all expected acknowledgements at $send\_tm$.

END_BROADCAST i → EndBrdcast i
→ **int** MyAnimator::End_Brdcast(**int** $start\_tm = i$)

At $start\_tm$ the end of the animation will be animated.

7.2.3.b The GHS Spanning Tree Algorithm

The animation of GHS-MST algorithm recognizes following events and adds to the class *MyAnimator* following event functions:

Wake_up i j → WakeUp i j
→ **int** MyAnimator::Wake_Up(**int** $id = i$, **int** $start\_tm = j$)

Process with identifier id wakes up at $start\_tm$.

```
Send_Mess_Name i j k → SendMess_Name i j k
→ int MyAnimator::Send_Mess_Name(int id = i, int adj = j,
                                 int start_tm = k)
```

$Mess\_Name$ denotes the name of a message chosen $\in$ {Initiate, Test, Connect, Report, Change-Root, Accept, Reject}. This message is sent from process identified by id to process identified by adj at $start\_tm$.

```
Rec_Mess_Name i j k → RecMess_Name i j k
→ int MyAnimator::Rec_Mess_Name(int id = i, int from = j,
                                int start_tm = k)
```

$Mess\_Name$ denotes the name of a message chosen $\in$ {Initiate, Test, Connect, Report, Change-Root, Accept, Reject}. This message is received from process identified by id from process identified by $from$ at $start\_tm$.

```
Next_Level i j → NextLevel i j
→ int MyAnimator::Next_Level(int id=i, int start_tm=j)
```

Process with identifier id starts the next level as a leader of a component at $start\_tm$.

```
END i j → End i j
→ int MyAnimator::End_Anim(int id = i, int start_tm = j)
```

Process with identifier id concluded from its convergecast information at $start\_tm$ that there exist no further outgoing edges. Thus it is the leader of the whole MST. The end of the animation will be animated.

7.2.3.c The Ricart and Agrawala Algorithm

The animation of the algorithm by Ricart and Agrawala recognizes following events and adds to the class *MyAnimator* following event functions:

```
TRYING i j → Trying i j
→ int MyAnimator::Trying(int id = i, int start_tm = j)
```

Process with identifier id becomes interested in its resources at $start\_tm$.

```
SEND_REQ i j k → SendReq i j k
→ int MyAnimator::Send_Req(int id = i, int adj = j, int start_tm = k)
```

Process with identifier id sends a message to process with identifier adj at $send\_tm$ in order to request access to a shared resource.

```
REC_REQ i j k → RecReq i j k
→ int MyAnimator::Rec_Req(int id = i, int from = j, int start_tm = k)
```

Process with identifier id receives a *request* message from process with identifier $from$ at $start\_tm$.

`SEND_ACK i j k → SendAck i j k`
`→` **int** `MyAnimator::Send_Ack(`**int** `id,` **int** `to,` **int** `start_tm)`

Process with identifier id acknowledges process with identifier to access to a shared resource at $send\_tm$.

`REC_ACK i j k → RecAck i j k`
`→` **int** `MyAnimator::Rec_Ack(`**int** $id = i$`,`**int** $from = j$`,`**int** $start\_tm = k$`)`

Process with identifier id receives an acknowledgement from process with identifier $from$ at $send\_tm$.

`USE i j → Use i j`
`→` **int** `MyAnimator::Use(`**int** $id = i$`,` **int** $start\_tm = j$`)`

Process with identifier id enters the critical section at $start\_tm$.

`EXIT_CS i j → ExitCS i j`
`→` **int** `MyAnimator::ExitCS(`**int** $id = i$`,` **int** $start\_tm = j$`)`

Process with identifier id exits the critical section at $start\_tm$.

`SLEEPING i j → Sleeping i j`
`→` **int** `MyAnimator::Sleeping(`**int** $id = i$`,` **int** $start\_tm = j$`)`

Process with identifier id changes its state to sleeping at $start\_tm$.

`END i → EndAnim i`
`→` **int** `MyAnimator::End_Anim(`**int** $start\_tm = i$`)`

The animation ends at $start\_tm$.

7.2.3.d The Chandy and Mistra Algorithm

The animation of the algorithm by Chandy and Mistra recognizes following events and adds to the class *MyAnimator* following event functions:

`START_COL i j → StartCol i j`
`→` **int** `MyAnimator::Start_Col(`**int** $id = i$`,` **int** $start\_tm = j$`)`

Process with identifier id starts with the colouring algorithm at $start\_tm$.

`DEC_COL i j → DecCol i j`
`→` **int** `MyAnimator::Dec_Col(`**int** $id = i$`,` **int** $start\_tm = j$`)`

Process with identifier id decided for a colour at $start\_tm$.

`TAKE_FORK i j k → TakeFork i j k`
`→` **int** `MyAnimator::Take_Fork(`**int** $id = i$`,`**int** $adj = j$`,`**int** $start\_tm = k$`)`

At $start\_tm$ process with identifier id takes the fork which is shared with process

identified by adj.

`HUNGRY i j` $\rightarrow$ `Hungry i j`
$\rightarrow$ **int** `MyAnimator::Trying(`**int** $id = i$, **int** $start\_tm = j$`)`

Process with identifier id gets interested in its resources at $start\_tm$.

`SEND_REQ i j k` $\rightarrow$ `SendReq i j k`
$\rightarrow$ **int** `MyAnimator::Send_Req(`**int** $id = i$, **int** $adj = j$, **int** $start\_tm = k$`)`

At $start\_tm$ process with identifier id sends a *request* message to process with identifier adj.

`REC_REQ i j k` $\rightarrow$ `RecReq i j k`
$\rightarrow$ **int** `MyAnimator::Rec_Req(`**int** $id = i$, **int** $from = j$, **int** $start\_tm = k$`)`

At $start\_tm$ process with identifier id receives a *request* message from process with identifier adj.

`SEND_FORK i j k` $\rightarrow$ `SendFork i j k`
$\rightarrow$ **int** `MyAnimator::Send_Req(`**int** $id = i$, **int** $adj = j$, **int** $start\_tm = k$`)`

At $start\_tm$ process with identifier id sends a message *fork* to process with identifier adj. The message signals the receiver that it can access a shared resource.

`REC_FORK i j k` $\rightarrow$ `RecFork i j k`
$\rightarrow$ **int** `MyAnimator::Rec_Fork(`**int** $id = i$, **int** $from = j$, **int** $start\_tm = k$`)`

At $start\_tm$ process with identifier id receives a message *fork* from process with identifier adj.

`USE i j` $\rightarrow$ `Use i j`
$\rightarrow$ **int** `MyAnimator::Use(`**int** $id = i$, **int** $start\_tm = j$`)`

Process with identifier id enters its critical section at $start\_tm$.

`EXIT_CS i j` $\rightarrow$ `ExitCS i j`
$\rightarrow$ **int** `MyAnimator::ExitCS(`**int** $id = i$, **int** $start\_tm = j$`)`

Process with identifier id exits its critical section at $start\_tm$.

`SLEEPING i j` $\rightarrow$ `Sleeping i j`
$\rightarrow$ **int** `MyAnimator::Trying(`**int** $id = i$, **int** $start\_tm = j$`)`

Process with identifier id changes its state to sleeping at $start\_tm$.

`END i` $\rightarrow$ `EndAnim i`
$\rightarrow$ **int** `MyAnimator::End_Anim(`**int** $start\_tm = i$`)`

At $start\_tm$ the end of the animation will be animated.

7.2.3.e The Choy and Singh Alg. with Failure Locality δ

This animation uses a further view called the state window of the basic view (see also chapter 5.8). This view will be introduced first before proceeding to the events and event functions recognized by the main program and added to class *MyAnimator*. The following structure was applied to class *BasicView*:

```
class BasicView:public View
{
public:
   BasicView();
   void Init(int, int, double, double);
   void Wait1(int, int, int);
   void Wait2(int, int, int);
   void Collect(int, int, int);
   void Use_CS(int, int, int);
   void Sleeping(int, int, int);

private:
   ...
};
```

void `BasicView::Init(`**int** `send_tm,` **int** `max_proc,` **double** `ixrad,`
double `iyrad)`

The state window of the basic view will appear on screen at $send\_tm$. Hereby parameter max_proc denotes the maximum umber of processes which will be used. The size of the circles representing a process is given by parameters $ixrad$ and $iyrad$.

void `BasicView::Wait1(`**int** `send_tm,` **int** `id,` **int** `frametime)`

A process with identifier id will appear at $send\_tm$. It will continuously be cycling in the area which indicates that this process is in state *wait1*. Parameter $frametime$ is used to delete old actions and animation objects.

void `BasicView::Wait2(`**int** `send_tm,` **int** `id,` **int** `frametime)`

At $send\_tm$ the process with identifier id will move to the area which indicates that this process is in state *wait2*.

void `BasicView::Collect(`**int** `send_tm,` **int** `id,` **int** `frametime)`

At $send\_tm$ the process with identifier id will move to the area which indicates that this process is in state *collect*.

void `BasicView::Use_CS(`**int** `send_tm,` **int** `id,` **int** `frametime)`

At $send\_tm$ the process with identifier id will move to the area which indicates that this process is inside its critical section.

void `BasicView::Sleeping(`**int** `send_tm, ` **int** `id, ` **int** `frametime)`

Process with identifier id will vanish at $send\_tm$.

events: The animation recognizes the following events and adds to the class *MyAnimator* the following event functions:

`START_COL i j → StartCol i j`
→ **int** `MyAnimator::Start_Col(`**int** $id = i$, **int** $start\_tm = j$`)`

Process with identifier id starts with the colouring algorithm at $start\_tm$.

`DEC_COL i j → DecCol i j`
→ **int** `MyAnimator::Dec_Col(`**int** $id = i$, **int** $start\_tm = j$`)`

Process with identifier id decided for a colour at $start\_tm$.

`TAKE_FORK i j k → TakeFork i j k`
→ **int** `MyAnimator::Take_Fork(`**int** $id = i$, **int** $adj = j$, **int** $start\_tm = k$`)`

At $start\_tm$ process with identifier id takes the fork which is shared with process identified by adj.

`WAIT1 i j → Wait1 i j`
→ **int** `MyAnimator::Wait1(`**int** $id = i$, **int** $start\_tm = j$`)`

At $start\_tm$ process with identifier id got interested in its resources and tries to enter the asynchronous doorway.

`WAIT2 i j → Wait2 i j`
→ **int** `MyAnimator::Wait2(`**int** $id = i$, **int** $start\_tm = j$`)`

At $start\_tm$ process with identifier id passed the asynchronous doorway and tries to enter the synchronous doorway.

`COLLECT i j → Collect i j`
→ **int** `MyAnimator::Collect(`**int** $id = i$, **int** $start\_tm = j$`)`

At $start\_tm$ process with identifier id exited the asynchronous doorway and tries to collect all forks.

`SEND_FORK i j k → SendFork i j k`
→ **int** `MyAnimator::Send_Fork(`**int** $id = i$, **int** $adj = j$, **int** $start\_tm = k$`)`

At $start\_tm$ process with identifier id sends a message *fork* to process with identifier adj. The message signals the receiver that it can access a shared resource.

`REC_FORK i j k → RecFork i j k`
→ **int** `MyAnimator::Rec_Fork(`**int** $id = i$, **int** $from = j$, **int** $start\_tm = k$`)`

At $start\_tm$ process with identifier id receives a message *fork* from process with iden-

tifier $from$.

```
SEND_BRDCAST i j k l → SendBrdcast i j k l
```
→ **int** `MyAnimator::Send_Brdcast(`**int** $id = i$`,` **int** $adj = j$`,`
int $m\_num = k$`,` **int** $start\_tm = l$`)`

At $start\_tm$ process with identifier id broadcasts message m_{m_num} to process with identifier adj.

```
REC_BRDCAST i j k l → RecBrdcast i j k l
```
→ **int** `MyAnimator::Rec_Brdcast(`**int** $id = i$`,` **int** $from = j$`,`
int $m\_num = l$`,` **int** $start\_tm = k$`)`

At $start\_tm$ process with identifier id receives message m_{m_num} from process with identifier $from$.

```
USE i j → Use i j
```
→ **int** `MyAnimator::Use(`**int** $id = i$`,` **int** $start\_tm = j$`)`

Process with identifier id enters its critical section at $start\_tm$.

```
EXIT_CS i j → ExitCS i j
```
→ **int** `MyAnimator::ExitCS(`**int** $id = i$`,` **int** $start\_tm = j$`)`

Process with identifier id exits its critical section at $start\_tm$.

```
SLEEPING i j → Sleeping i j
```
→ **int** `MyAnimator::Trying(`**int** $id = i$`,` **int** $start\_tm = j$`)`

Process with identifier id changes its state to sleeping at $start\_tm$.

```
END i → EndAnim i
```
→ **int** `MyAnimator::End_Anim(`**int** $start\_tm = i$`)`

At $start\_tm$ the end of the animation will be animated.

7.2.3.f The Choy and Singh Alg. with Failure Locality 4

The animation uses also the state view of the basic view and algorithm events described for the animation of Choy and Singh's algorithm with failure locality δ. The following events and event functions were added to the main program and class *MyAnimator*.

```
RELEASE i j → Release i j
```
→ **int** `MyAnimator::Release(`**int** $id = i$`,` **int** $start\_tm = j$`)`

Process with identifier id releases its higher coloured forks at $start\_tm$.

```
GOT_L_FORKS i j → GotLForks i j
```
→ **int** `MyAnimator::Got_L_Forks(`**int** $id = i$`,` **int** $start\_tm = j$`)`

Process with identifier id collected all its higher coloured forks at $start\_tm$.

`SEND_REQUEST i j k` $\rightarrow$ `SendReq i j k`
$\rightarrow$ **int** `MyAnimator::Send_Req(`**int** $id = i$, **int** $adj = j$, **int** $start\_tm = k$`)`

At $start\_tm$ process with identifier id sends a *request* message to process with identifier adj.

`REC_REQUEST i j k` $\rightarrow$ `RecReq i j k`
$\rightarrow$ **int** `MyAnimator::Rec_Req(`**int** $id = i$, **int** $from = j$, **int** $start\_tm = k$`)`

At $start\_tm$ process with identifier id receives a *request* message from process with identifier $from$.

7.2.3.g The Periodic Counting Network

There exist two different animations for the periodic counting network. The first animation reads algorithmic events from a trace-file, while the other interacts with the user by associating with inputs and processes of the counting network call-back routines. However, the interactive animation uses the same event functions as the trace-file animation, so only a single description for both animations will follow.

Besides the basic view the animations of counting networks show two further views for input and output of the counting network, both are implemented in two further classes (see also chapter 6.5). These classes are introduced before proceeding to describe events added to class *MyAnimator*.

```
class Input : public View
{
public:
  int Init(int, int);
  int Appear(int, int);
};
```

int `Input::Init(`**int** `frametime, int n)`

At $frametime$ the input window for a counting network of width n will be created.

int `Input::Appear(`**int** `channel, int frametime)`

At $frametime$ for input $chanel$ the displayed number of tokens is increased by one. An arrow marks the channel were the input token appears.

```
class Output : public View
{
public:
  int Init(int, int, double);
  int Vanish1(int);
  int Vanish2(int, int, int);
};
```

int `Output::Init(`**int** `frametime, int n, double rad)`

At $frametime$ the output window for a counting network of width n will be created. Tokens will be displayed with radius rad.

int `Output::Vanish1(`**int** `frametime)`

If at every output node a token is shown then at $frametime$ for every output a token will disappear.

int `Output::Vanish2(`**int** `frametime, int chanel, int number)`

At $frametime$ a token with identifier $number$ will appear on the output indicated by parameter $chanel$.

events: The animation recognizes the following events and adds to the class *MyAnimator* the following event functions:

`APPEAR i` $\rightarrow$ `Appear i`
$\rightarrow$ **int** `MyAnimator::Appear(`**int** $chanel = i$`)`

A token appears at input $chanel$ of the counting network.

`RECEIVE i j k l` $\rightarrow$ `Receive i j k l`
$\rightarrow$ **int** `MyAnimator::Receive(`**int** $block = i$`,` **int** $depth = j$`,`
int $number = k$`,` **int** $position = l$`)`

The process identified by parameters $block$, $depth$ and $number$ receives a token from the input channel defined by parameter $position$.

`SEND i j k l` $\rightarrow$ `Send i j k l`
$\rightarrow$ **int** `MyAnimator::Send(`**int** $block = i$`,` **int** $depth = j$`,` **int** $number = k$`)`

The process identified by parameters $block$, $depth$ and $number$ sends a token to its next output channel.

`VANISH i` $\rightarrow$ `Vanish i`
$\rightarrow$ **int** `MyAnimator::Vanish(`**int** $chanel = i$`)`

A token disappears at output $chanel$ of the counting network.

7.3 Algorithm Implementations Using DIAS

The algorithms were implemented for the simulator of LYDIAN. For each algorithm states and messages will be listed. Further transitions and the invoked functions will be described.

7.3.1 The Broadcast with Acknowledgement Algorithm

Below states, messages and transitions of the broadcast with acknowledgement algorithm are listed. Figure 7.4 shows the state diagram of the algorithm.

states: `sleeping, waiting`
messages: `BROADCAST, ACKNOW`

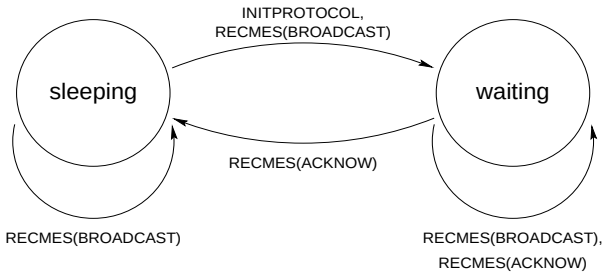

Figure 7.4: State diagram of the protocol

transitions:

`sleeping`	×	`RECMES(BROADCAST)`	→	`forward1`
`sleeping`	×	`RECMES(ACKNOW)`	→	`error`
`sleeping`	×	`INITPROTOCOL`	→	`start`
`waiting`	×	`RECMES(BROADCAST)`	→	`forward`
`waiting`	×	`RECMES(ACKNOW)`	→	`forward`
`waiting`	×	`INITPROTOCOL`	→	`error`

`start()`

The process which calls this function is initiator of the broadcast algorithm. If there exist any neighbours it will send *broadcast* messages and changes its state to *waiting*. Otherwise it terminates.

`forward()`

The process which calls this function is assumed to be in state *waiting* and it has received either message *broadcast* or an acknowledgement. A *broadcast* message will be responded with sending an acknowledgement. If the process received all acknowledgements it sends an acknowledgement to its parent process (the process which sent the first *broadcast* message) and terminates.

`forward1()`

The process which calls this function is in state *sleeping*. It has received a *broadcast* message from another process which will be remembered as the parent process. If there exist any other adjacent processes it will change to state *waiting* and send these processes *broadcast* messages. Otherwise it will answer the parent process with an acknowledgement and will terminate.

7.3.2 The GHS Spanning Tree Algorithm

Below states, messages and transitions of the GHS spanning tree algorithm are listed. Figure 7.5 shows the state diagram of the algorithm.

states: `sleeping, find, found`
messages: `CONNECT, INITIATE, TEST, ACCEPT, REJECT, REPORT, CHANGEROOT`

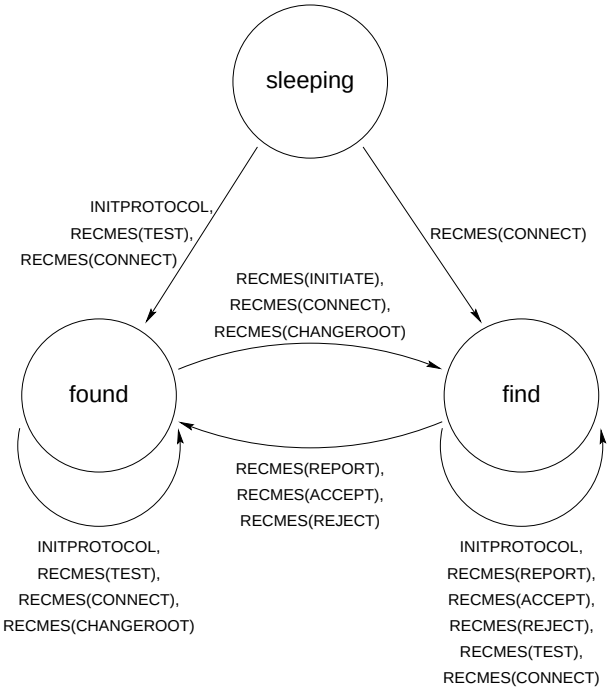

Figure 7.5: State diagram of the protocol

transitions:

```
sleeping  ×   RECMES(CONNECT)   →   wake_up_0();
                                    resp_connect()
sleeping  ×  RECMES(INITIATE)   →       error
sleeping  ×    RECMES(TEST)     →   wake_up_0();
                                    resp_test()
sleeping  ×   RECMES(ACCEPT)    →       error
sleeping  ×   RECMES(REJECT)    →       error
sleeping  ×   RECMES(REPORT)    →       error
sleeping  × RECMES(CHANGEROOT)  →       error
sleeping  ×    INITPROTOCOL     →    wake_up_0()
  find    ×   RECMES(CONNECT)   →  resp_connect()
  find    ×  RECMES(INITIATE)   →  resp_initiate()
  find    ×    RECMES(TEST)     →   resp_test()
  find    ×   RECMES(ACCEPT)    →  resp_accept()
  find    ×   RECMES(REJECT)    →  resp_reject()
  find    ×   RECMES(REPORT)    →  resp_report()
```

```
 find    × RECMES(CHANGEROOT) → resp_change_root()
 find    ×    INITPROTOCOL    →      no call
found    ×  RECMES(CONNECT)   →  resp_connect()
found    ×  RECMES(INITIATE)  →  resp_initiate()
found    ×    RECMES(TEST)    →    resp_test()
found    ×   RECMES(ACCEPT)   →   resp_accept()
found    ×   RECMES(REJECT)   →   resp_reject()
found    ×   RECMES(REPORT)   →   resp_report()
found    × RECMES(CHANGEROOT) → resp_change_root()
found    ×    INITPROTOCOL    →      no call
```

`wake_up_0()`

The process which calls this function is the leader of a level 0 component. It tries to connect with its local MWOE. If there does not exist any neighbours the process finished the computation of the MWOE and hence it will terminate.

`resp_connect()`

The process which calls this function received a *connect* message. Depending on the level of the sender's component and the state of the link where the message was received the process will absorb the component, delay its decision or start as the leader of the merged components.

`resp_initiate()`

The process which calls this function received an *initiate* message. It updates its values for UID, level and best weight. Further it remembers the sender as parent process. Delayed components which sent a *connect* message before will be marked as tree edges. Then, the process sends *initiate* messages to tree edges with exception of the parent process. Moreover it tests its minimum non tree edge by also taking into account possible delayed *test* messages.. If the process has already determined its local MWOE it will change its state to *found* and *report* the result to its parent. Otherwise the state of the process will be *find* and the process expects the reports of its tree edges and the result of the tested edge.

`resp_test()`

A *test* message was received. Depending on level and UID of the sender the message will be delayed or the message will be replied with *accept* (processes belong to different components) or *reject* (processes belong to the same component).

`resp_accept()`

An *accept* message was received. The process can conclude that the sender belongs to another component and thus can update its best edge. If all expected *report* messages are received the process will change its state to *found*. In case it is not the component's leader it reports its parent node the weight of its best edge. Otherwise it will search for a new leader if a MWOE of the component exists. If there is not any MWOE the MST

is computed and the process terminates.

`resp_reject()`

A *reject* message was received. The process must determine a new minimum non tree edge to be tested. If such an edge does not exist it follows the same scenario as described in function *resp_accept()* which results in either reporting to the parent or searching for a new leader or terminating the algorithm.

`resp_report()`

The process received message *report*. It updates the value of the best edge and follows the scenario described in function *resp_accept()* if all other expected reports have arrived and a non tree edge was tested. Then the process either reports to the parent process or determines a new leader or terminates the algorithm.

`resp_change_root()`

The process received a message *change-root*. If its best edge is a tree edge it will send message change root along the respective link, otherwise it will try to combine with the other component. If this process has received a *connect* message before and its id is larger than the id of the adjacent process it will start as the leader of the merged component and will send *initiate* messages along its tree edges. In the other case it will send message *connect*.

7.3.3 The Ricart and Agrawala Algorithm

The implemented algorithm algorithm allows the user to specify the number of entries to the critical section and to define limits for accessing a resource and remaining in state sleeping. The timing limits are used together with a timeout event in order to simulate the period of time in which a process remains in either of these states. Below states, messages and transitions of the resource allocation algorithm by Ricart and Agrawala are listed. Figure 7.6 shows the state diagram of the algorithm.

states: `sleeping, trying, use`
messages: `REQUEST, ACKNOW`

transitions:

sleeping	×	RECMES(REQUEST)	→	receiped()
sleeping	×	RECMES(ACKNOW)	→	error
sleeping	×	INITPROTOCOL	→	start()
sleeping	×	TIMEOUT_1	→	request()
sleeping	×	TIMEOUT_2	→	error
trying	×	RECMES(REQUEST)	→	receiped()
trying	×	RECMES(ACKNOW)	→	receiped()
trying	×	INITPROTOCOL	→	error
trying	×	TIMEOUT_1	→	error
trying	×	TIMEOUT_2	→	error
use	×	RECMES(REQUEST)	→	receiped()

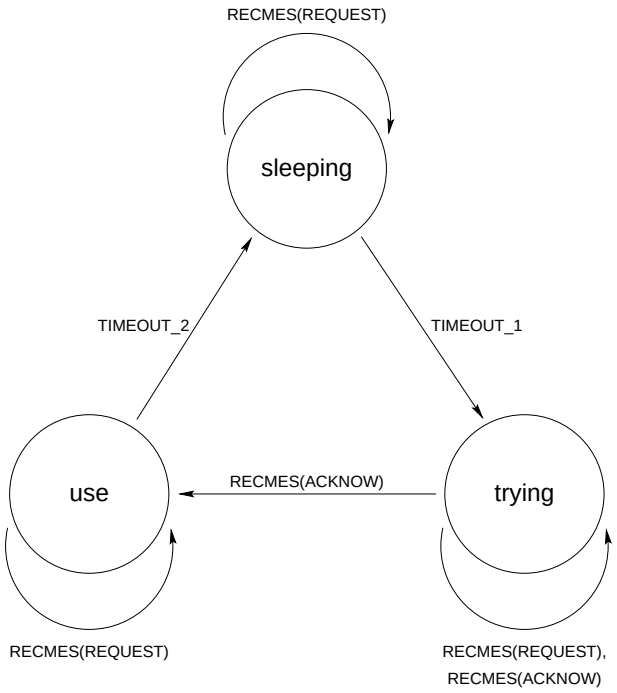

Figure 7.6: State diagram of the protocol

use	×	RECMES(ACKNOW)	→	error
use	×	INITPROTOCOL	→	error
use	×	TIMEOUT_1	→	error
use	×	TIMEOUT_2	→	exit_CS()

`start()`

The process was waken up. It initializes a randomized time for starting a timer. The values are chosen within the limits specified by the user. When the process receives a timeout event it will try to access its critical section.

`receiped()`

- *case* 1*:* The process received a *request* message. Depending on its state and local clock it will either acknowledge the sender to access the common resource or it will delay the sender.
- *case* 2*:* The process received an acknowledgement. If this acknowledgement is the last expected acknowledgement it will access its critical section. The process will change its state to *use* and initialize a randomized time for starting a timer. The limits will be chosen within the limits specified by the user. The timeout event will mark the exit of the critical section.

`request()`

The process became interested in its resources. Hence it sends *request* messages with

a stamp of its local clock to all adjacent processes. The new state of the process is *trying*.

`exit_CS()`

The process exits its critical section, so it changes its state to *sleeping* and sends acknowledgements to all processes which requested resources. Further it initializes a randomized time for starting a timer. The value is chosen within the user specified limits. The timeout event will cause the process to become interested in its resources again later.

7.3.4 $\delta + 1$ Colouring by Luby

This algorithm was implemented such that it can invoke another algorithm. A process which finished with the colouring algorithm will call a function *start_alg()*. Below states, messages and transitions of the $\delta + 1$ colouring algorithm by Luby are listed. Figure 7.7 shows the state diagram of the algorithm.

states: `sleeping, waiting`
messages: `COLOR, CONFIRM`

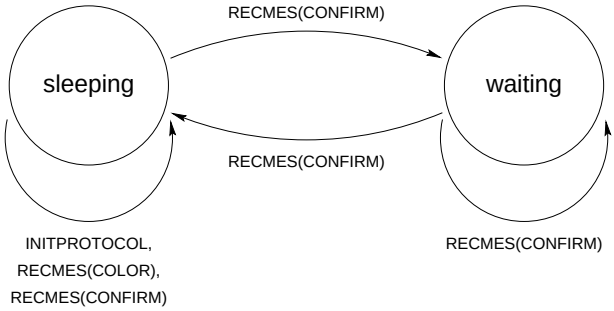

Figure 7.7: State diagram of the protocol

transitions:

```
sleeping  ×   RECMES(COLOR)    →  col_receipe()
sleeping  ×  RECMES(CONFIRM)   →  col_receipe()
sleeping  ×   INITPROTOCOL     →   col_start()
 waiting  ×   RECMES(COLOR)    →      error
 waiting  ×  RECMES(CONFIRM)   →  col_receipe()
 waiting  ×   INITPROTOCOL     →      error
```

`col_start()`

If a temporary colour is not chosen the process will choose an available value. The process sends a *colour* message including the value of the temporary colour to all adjacent processes which have not decided for a colour. If from all expected neighbours a *colour* message is received the process will create a *confirm* message. This message will be sent to all adjacent processes if the final colour value is > 0. Otherwise the

message will only be sent to those processes which participated searching a new colour in this phase.

`col_receipe()`

- *case process received a* colour *message:* If a colour is not chosen the process will choose an available value. It compares the value of the sender with its own value. If they are equal the final colour of the process will be set to zero. If this message is the last expected *colour* message, *confirm* messages will be sent as described for function *col_start()*.

- *case process received a* confirm *message in state waiting:* The process has decided for its colour but still expects colours from its neighbours. It updates the colour value for the sender of the message. In case the message is the last expected *confirm* message the process will change its state to sleeping and start with the next algorithm.

- *case process received a* confirm *message in state sleeping:* If the sender's colour is greater than zero the value will be removed from process's palette of available colours. If all expected *confirm* messages were sent the process will either finish the algorithm (case all processes decided for a colour with value greater than zero), it will change to state waiting (case its final colour is greater zero) or it will start with the next phase of the algorithm by calling function *col_start()*.

7.3.5 The Chandy and Mistra Algorithm

The implementation of the algorithm by Chandy and Mistra first starts the $\delta+1$ colouring by Luby in order to initialize the precedence graph. Analogous to the algorithm by Ricart and Agrawala the user is able to specify the number of entries to the critical section and to define limits for accessing a resource and remaining in state sleeping. The timing limits are used together with a timeout event in order to simulate the period of time in which a process remains in either of these states. Below states, messages and transitions of the resource allocation algorithm by Chandy and Mistra are listed. Hereby the states of the colouring phase are omitted. Figure 7.8 shows the state diagram of the algorithm.

states: `sleeping, hungry, eating`
messages: `FORK, REQUEST`

transitions:

sleeping	×	RECMES(FORK)	→	error
sleeping	×	RECMES(REQUEST)	→	receiped()
sleeping	×	INITPROTOCOL	→	colouring
sleeping	×	TIMEOUT_1	→	wake_up()
sleeping	×	TIMEOUT_2	→	error
hungry	×	RECMES(FORK)	→	receiped()
hungry	×	RECMES(REQUEST)	→	receiped()
hungry	×	INITPROTOCOL	→	error
hungry	×	TIMEOUT_1	→	error

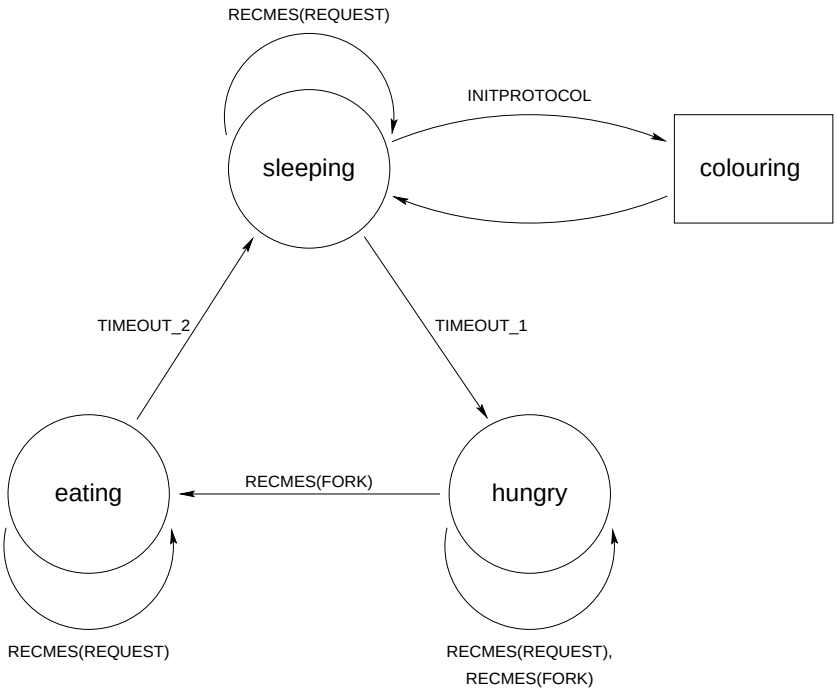

Figure 7.8: State diagram of the protocol

`hungry`	×	`TIMEOUT_2`	→	`error`
`eating`	×	`RECMES(FORK)`	→	`error`
`eating`	×	`RECMES(REQUEST)`	→	`receiped()`
`eating`	×	`INITPROTOCOL`	→	`error`
`eating`	×	`TIMEOUT_1`	→	`error`
`eating`	×	`TIMEOUT_2`	→	`exit_CS()`

`wake_up()`

The process became interested in its resources due to a timeout event. If the process owns all forks it will switch to state *hungry* and access its resources. The time that the process will spent inside its critical section is given by a timer which will be randomly chosen within the user specified timing assumptions. Otherwise (in case the process does not own its resources) the process will sent *request* messages to its neighbours.

`receiped()`

- *case received request:* If the process holds a dirty fork and is not inside its critical section, it will reply by sending message *fork*. Otherwise it will delay the response until it could exit its critical section.

- *case received fork:* If all expected forks are received the process will switch its state to *hungry* and access its resources. The process initializes a timer according to the user specified timing assumptions. This defines the period a process will stay inside its critical section.

`exit_CS()`

The process exits its critical section due to a timeout event. It changes its state to sleeping and releases all requested forks. Further it initializes a timer according to the user specified timing assumptions. This defines the period a process will remain not being interested in its resources.

7.3.6 The Choy and Singh Algorithm with Failure Locality δ

Like the implementation of the algorithm by Chandy Mistra the implementation starts the $\delta + 1$ colouring by Luby in order to initialize the precedence graph. Also the user is able to specify the number of entries to the critical section and the limits for accessing a resource and remaining in state sleeping. The timing limits are used together with a timeout event in order to simulate the period of time in which a process remains in either of these states. Below states, messages and transitions of the δ fault tolerant resource allocation algorithm by Choy and Singh are listed. Hereby the states of the colouring phase are omitted. Figure 7.9 shows the state diagram of the algorithm.

states: `sleeping, wait1, wait2, collect, eating`
messages: `FORK, BROADCAST`

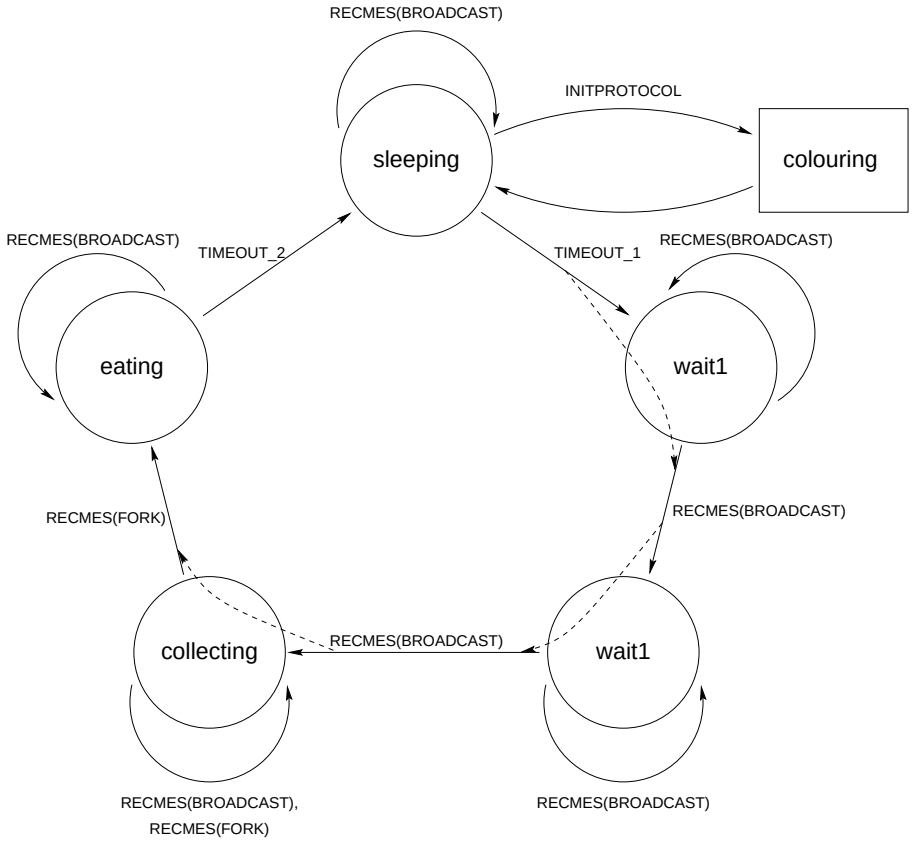

Figure 7.9: State diagram of the protocol

transitions:

sleeping	×	RECMES(FORK)	→	error
sleeping	×	RECMES(BROADCAST)	→	receiped()
sleeping	×	INITPROTOCOL	→	colouring
sleeping	×	TIMEOUT_1	→	wake_up()
sleeping	×	TIMEOUT_2	→	error
wait1	×	RECMES(FORK)	→	error
wait1	×	RECMES(BROADCAST)	→	receiped()
wait1	×	INITPROTOCOL	→	error
wait1	×	TIMEOUT_1	→	error
wait1	×	TIMEOUT_2	→	error
wait2	×	RECMES(FORK)	→	error
wait2	×	RECMES(BROADCAST)	→	receiped()
wait2	×	INITPROTOCOL	→	error
wait2	×	TIMEOUT_1	→	error
wait2	×	TIMEOUT_2	→	error
collect	×	RECMES(FORK)	→	receiped()
collect	×	RECMES(BROADCAST)	→	receiped()
collect	×	INITPROTOCOL	→	error
collect	×	TIMEOUT_1	→	error
collect	×	TIMEOUT_2	→	error
eating	×	RECMES(FORK)	→	error
eating	×	RECMES(BROADCAST)	→	receiped()
eating	×	INITPROTOCOL	→	error
eating	×	TIMEOUT_1	→	error
eating	×	TIMEOUT_2	→	exit_CS

`pass_1st_door()`

A process calls this function when it was allowed to pass the asynchronous doorway. It changes its state to *wait2* and checks for also passing the synchronous doorway. If it may pass the synchronous doorway as well it will broadcast m_2 to all neighbours and it will call function *pass_2nd_door()*.

`pass_2nd_door()`

The process calls this function when it was allowed to pass the synchronous doorway. It changes its state to *wait2* and checks for forks. If all forks are collected it will broadcast m_3 to lower coloured neighbours and will call function *eat()*.

`eat()`

The process changes its state to *eating* and initializes a timer according to the limits specified by the user. The timer defines the duration of the process accessing its critical section.

`wake_up()`

The process became interested in its resources due to a timeout event and checks for entering the asynchronous doorway. If the process may enter the asynchronous doorway it will broadcast m_1 to all higher coloured neighbours. Afterwards it calls function $pass\_1st\_door()$.

`receiped()`

- *case process received a broadcast message:* The process updates the status of its doorways according to the broadcasted value $\in m_1, m_2, m_3$. If the process wishes to pass a doorway and the new evaluated state allows passing this doorway, the process will execute the doorway entry code and call the accordant function *pass_1st_dooray()* or *pass_2nd_dooray()*. If value m_2 is broadcasted then the sender also requests the common fork. It will reply with sending a fork in states *sleeping, wait1, wait2*. In state *collect* a fork will only be sent if the sender has a lower colour value, while in state *eating* never forks will be sent.
- *case process received a fork message:* If the process received all expected *fork* messages it will broadcast m_3 to all lower coloured neighbours and call function *eat()*.

`exit_CS()`

The process exits its critical section due to a timeout event. It changes its state to sleeping and releases all requested forks. Further it initializes a timer according to the user specified timing assumptions. This defines the period a process will remain not being interested in its resources.

7.3.7 The Choy and Singh Algorithm with Failure Locality 4

Only some functions differ from previous implementation in section 7.3.6, but the function names stay the same. Therefore only new transitions and changes of functions will be described in the following.

states: `sleeping, wait1, wait2, collect, eating`
messages: `FORK, BROADCAST, REQUEST`

transitions:

sleeping	×	RECMES(REQUEST)	→	receiped()
wait1	×	RECMES(REQUEST)	→	receiped()
wait2	×	RECMES(REQUEST)	→	receiped()
collect	×	RECMES(REQUEST)	→	receiped()
eating	×	RECMES(REQUEST)	→	receiped()

`pass_2nd_door()`

The process changes its state to *collect* and distinguishes between the following cases:

- *case process collected all forks:* The process broadcasts m_3 to all lower coloured neighbours and calls function *eat()*.

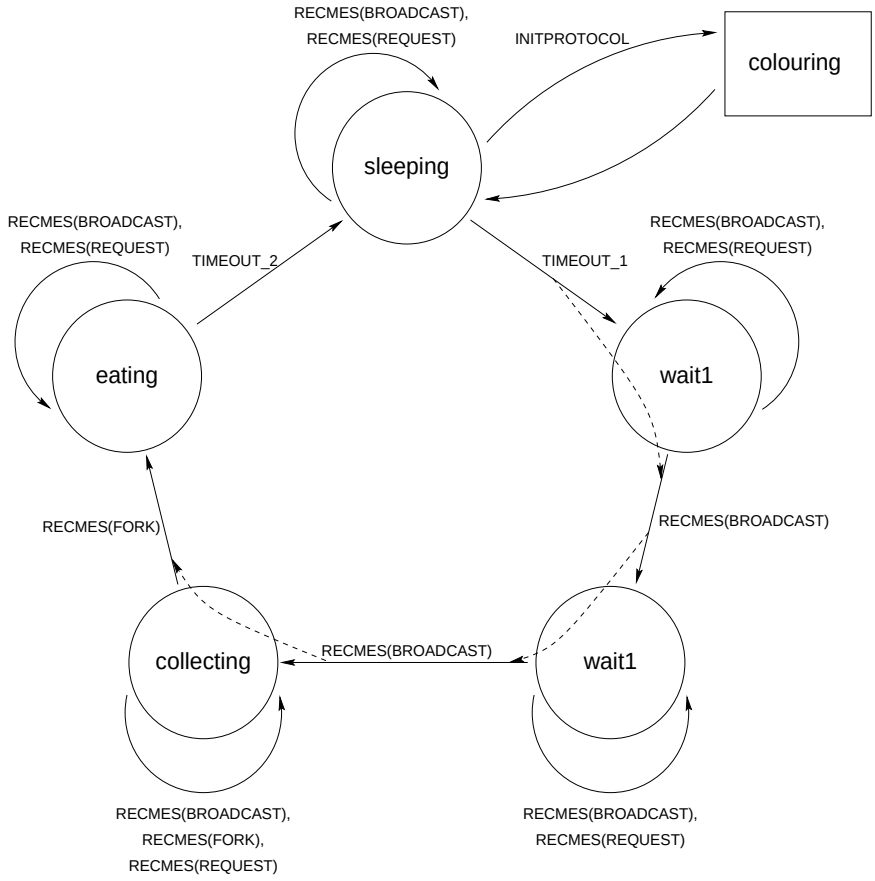

Figure 7.10: State diagram of the protocol

- *case process received all forks from lower coloured neighbours:* Process sends *request* messages to higher coloured processes.
- *case process did not receive all forks from lower coloured neighbours:* Process sends first *request* messages to lower coloured processes

`receiped()`

- *case process received a broadcast message:* The only difference in this case is caused by not interpreting m_2 as a *request* message.
- *case process received a request message:* In states *sleeping, wait1, wait2* the process replies with sending message *fork*. In state *collect* it will send a fork only if the sender has lower colour or it has not collected all lower coloured forks. When a process in state *collect* sends a *fork* message to a process with lower colour, a value will be included signaling the receiver that the fork is also desired by the sender. If the process has collected all lower coloured forks until releasing one it will also have to release all collected higher coloured forks which were requested before. In state *eating* the process will never release a fork. If forks are not released they will be marked as being requested.

- *case process received a fork message:* If the fork is the process's last expected fork it will broadcast m_3 to all lower coloured neighbours and call function *eat()*. If the fork is only the process's last expected lower coloured fork it will request its higher coloured forks. In case the received fork is also desired by its sender, but the process has not collected all lower coloured forks it will reply with sending message *fork* (important case since after sending a request another party might request a lower coloured fork).

7.3.8 The Periodic Counting Network

The following algorithm simulates the behaviour of a counting network. The user can specify limits for the number of tokens which are produced at each input of the counting network. The algorithm assumes that inside the communication graph processes are connected in the way balancers are connected in the description of the periodic counting network (see chapter 6.3). Below states, messages and transitions of this algorithm are listed.

states: `sleeping, produce, wait, consume`
messages: `PACKAGE, END`

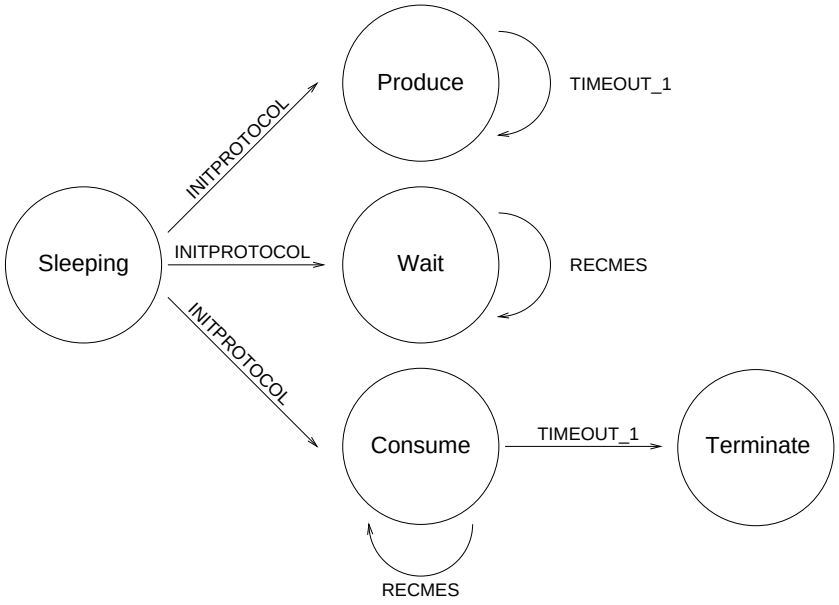

Figure 7.11: State diagram of the protocol

transitions:

```
sleeping  ×   RECMES(PACKAGE)   →     awake()
sleeping  ×     RECMES(END)     →      error
sleeping  ×    INITPROTOCOL     →     start()
sleeping  ×      TIMEOUT_1      →      error
 produce  ×   RECMES(PACKAGE)   →      error
 produce  ×     RECMES(END)     →      error
```

produce	×	INITPROTOCOL	→	error
produce	×	TIMEOUT_1	→	produce()
wait	×	RECMES(PACKAGE)	→	wait()
wait	×	RECMES(END)	→	wait()
wait	×	INITPROTOCOL	→	start()
wait	×	TIMEOUT_1	→	wait2()
consume	×	RECMES(PACKAGE)	→	consume()
consume	×	RECMES(END)	→	consume()
consume	×	INITPROTOCOL	→	start()
consume	×	TIMEOUT_1	→	terminate()

`start()`

The process is waken up. Depending whether the process is an input node of the network, a balancer or an output node of the network it changes its state to *produce*, *wait* or *consume*. If the process is in state *produce* it will initialize a timer. This defines the time this process must wait until it produces a token.

`produce()`

If the process has already produced all tokens it will send message *end* to its adjacent balancer in order to signal the topmost output process to terminate the program.

Otherwise the process produces a token and sends message *package* to its adjacent balancer. Further it initializes a timer in order to define the time this process must wait until it may produce the next token.

`wait()`

The process is in state *wait* and received either message *end* or message *package*. It buffers the message and initializes a timer. This defines the time this process must wait until it passes this message to the next balancer or output node.

`wait2()`

The process in state *wait* reads the next message from its buffer. A *end* message will be sent to the upper output channel such that it will finally reach the topmost output of the network, while a *package* message will be sent to the output channel to which the respective counting network would send the next token in the same situation.

`consume()`

The process is in state *consume* receives either a message *package* or a message *end*. If the process has received as many *end* messages as the width of the network then the process initializes a timer. This timer defines the time until the protocol terminates.

`awake()`

A sleeping process received a message. The process determines whether it is a balancer or an output node of the counting network and calls depending on its state either

function `wait()` or `consume()`.

`terminate()`

The process terminates the protocol.

7.4 Creation of Network Description Files

Network description files define the structure of how processes are interconnected. They define for each process p of the network adjacent processes which are directly connected with a link to p and also the quality of the link i.e how fast is a message expected to traverse this link. Further it can be specified when processes wake up and the time a process needs to execute a computation. The simulator will execute protocols according to the applied network description file. For the animations also the network description files are of importance since they allow users to design themselves a graph's layout.

The implementation uses an instance of LEDA's graphwin for creation of network description files. Hereby to the graphwin menus a topic *timing* was added. This allows to apply a time distribution to the created graph by keeping the whole functionality of graphwin e.g. showing the graph in different layouts. For writing the network description to a file the submenus *"Save Distribution"* and *"Save Graph Structure"* were added to the file menu of graphwin. The first one creates a network description file for the simulator, while the other is created for animation purposes.

The menu *timing* contains submenus which allow either the modification of the used distribution or gain information. Following submenus were implemented:

- *asynchronous timing:* The user is allowed to apply a timing distribution to processes and links which results in an asynchronous execution. The user can select among the following distributions:
  - *uniform distribution:* The value is chosen between an upper and a lower limit. Hereby each value is selected with same probability.
  - *geometric distribution:* The user specifies a lower and upper bound within a value is chosen. Hereby a third value called *mean* is the most likely value.
  - *normal distribution:* The user defines the upper/lower limit and the variance of the distribution.
  - *deterministic distribution:* The user selects one value which specifies the exact time.
- *synchronous timing:* The user is allowed to apply a timing distribution for synchronous executions. The user can select among the following distributions:
  - *ABD*
  - *absolute synchronous:* Each process is working at the same computation step at the same time.
- *information:* This submenu informs about the selected timing.

For each distribution the user has to specify values for process step delay, link delay and initialization time of processes. With a menu button called *options* the user can specify values which are only valid for user selected processes or links.

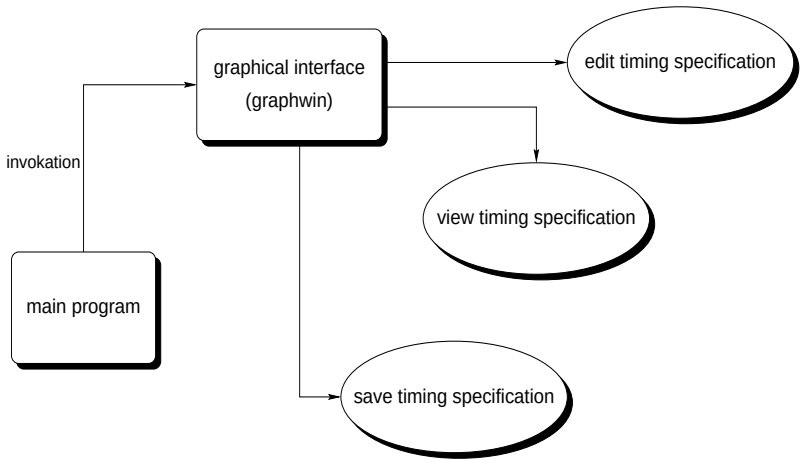

Figure 7.12: The flow of information of the interface for creating network description files

The user can change interactively some values which specify the timing. For this reason an object called *time_type* was created. A global variable *time_spec* of type *time_type* contains the latest timing information and provides functions for the output of data. Clicking submenus which change the timing information will call functions allowing the modification of *time_spec*. The flow of information can be seen in figure 7.12. Hereby the following functions support the interaction between the user and the graphical interface:

void `select_edge_param(GraphWin& gw,`
`time_type& new_time_spec)`

A menu will be created which allows the user to change the timing specification given by parameter new_time_spec for selected links of graphwin object gw.

void `select_some(GraphWin& gw, time_type& new_time_spec)`

A menu will be created which asks the user to specify the initialization times for selected processes of graphwin object gw. Parameter new_time_spec containing timing information for gw will be updated.

void `save_graph_str(GraphWin& gw)`

This function saves the graph structure of graphwin object gw to a file which can be read by an animator.

void `save_distr(GraphWin& gw)`

This function saves the timing information of global variable $time\_spec$ and graphwin object gw to a file which can be read by the simulator.

void `show_init_node(GraphWin& gw)`

For the graphwin object gw the initialization times of processes are shown.

void `show_com_edge(GraphWin& gw)`

For a selected link of graphwin object gw the communication times are displayed.

void `inform_timing(GraphWin& gw)`

The user will be informed about the timing specifications for graphwin object gw.

void `timing(GraphWin& gw,` **int** `dist)`

A menu appears which allows to modify the timing specifications of graphwin object gw according to the in parameter $dist$ chosen distribution.

`main()`

The main program creates a graphwin object called gw. It creates the respective menus for gw and initializes the appearance of gw. By calling for gw function *edit()* of the graphwin object the interaction with the user starts.

Chapter 8

Basic Findings and Future Work

This work offered a set of basic distributed algorithms and animations which are accessible inside LYDIAN. They can be tested with different communication graphs and timing assumptions selected by the user. All animations use the structure of animations which is introduced in chapter 2. The animations always show the basic view and offers the user to choose among a set of four further views each showing a specific aspect of the algorithm. The implementation of these four views can be used for any animation of a distributed algorithm using message passing. Therefore it will be easier to develop new animation for this class of algorithms since further developments need only to concentrate on the main idea of the algorithm.

The motivation of creating animations was to help an user understanding a particular algorithm. However, an animation cannot substitute studying also the theory of an algorithm. For this reason each introduced algorithm of this work was explained and analyzed.

The educational benefit of these animations needs to be examined in the future. This will be done for example in the course on distributed algorithms by Marina Papatriantafilou and Philippas Tsigas.

Besides the extension of algorithms and animations available for LYDIAN future work will be about building tools which allow users to create animations themselves. It might also be interesting to allow users to interact in an animation which reads events from a trace-file. The user might explore for a state of the system different scenarios which could convince the user from the correctness of the algorithm. Due to the interaction the following events of the trace-file might not lead to a consistent execution anymore. Hence, the problem of keeping the trace-file consistent needs a fast and efficient solution in order to guarantee a smooth animation.

Bibliography

[1] J. Aspnes, M. Herlihy, N. Shavit, *Counting Networks*, 1993.

[2] K.M. Chandy and J. Mistra. The Drinking Philosophers Problem. *ACM Transactions on Programming Languages and Systems*,633-646, 1984.

[3] M. Choy and A.K. Singh. Resource Allocation. *ACM Transactions on Programming Languages and Systems*, 535-559, May 1995.

[4] T.H. Corman, C.E. Leiserson, R.L. Rivest. *Introduction to Algorithms*, page 505, MIT Press, Cambridge, Massachusetts.

[5] R.G. Gallager, P.A. Humblet, and P.M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66-77, January 1983.

[6] D. Heller. *Motif Programming Manual for OSF/Motif Version 1.1*, O'Reilly & Associates, Sebastopol, CA, 1991.

[7] J.B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48-50, 1956.

[8] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558-565, July 1978.

[9] M. Luby. Removing Randomness in Parallel Computation Without a Processor Penalty. *9th Annual Symposium on Foundations of Computer Science*, 5:166-170, 1988.

[10] K. Mehlhorn, S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*, to appear with Cambridge University Press, 1999.

[11] N.A. Lynch. *Distributed Algorithms*, pages 63-70, 509-524, Morgan Kaufmann Publishers, Inc. San Francisco, California.

[12] M. Papatriantafilou P. Tsigas. Towards a Library of Distributed Algorithms and Animations. *Proceedings of the 4th International Conference on Computer Aided Learning and Instruction in Science and Engineering (CALISCE '98)*, pages 407-410, 1998.

[13] G. Ricart, A. K. Agrawala *Optimal Algorithm for Mutual Exclusion in Computer Networks, Communications of the ACM*, January 1981, Volume 24, Number 1, pp. 9-17.

[14] John Stasko. *POLKA Animation Designer's Package*, Technical Report, Georgia Institute of Technology, 1995.

[15] G. Tel *Introduction to Distributed Algorithms Cambridge University Press*, 1994.

[16] J.L. Welch, L. Lamport, N.A. Lynch. A lattice-structured proof technique applied to a minimum spanning tree algorithm. *In Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, page 28-43, Toronto, Ontario, Canada, August 1988.