Diplomarbeit:
Animations and Analysis of Distributed Algorithms

Boris Koldehofe

6th May 1999

Contents

Chapter 1

Introduction

Distributed algorithms are by nature difficult to understand due to the existence of asynchronous threads of control which interact 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 the main concepts which were 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 analyzed showing 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 to specify timing assumptions for the network of processes which is represented by the constructed graph.

In order to built the animations themselves, a library called POLKA [14] was 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*).

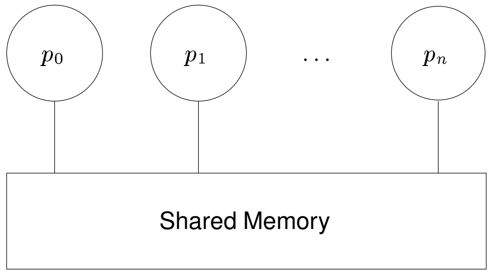

Figure 1.1: a shared memory model

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

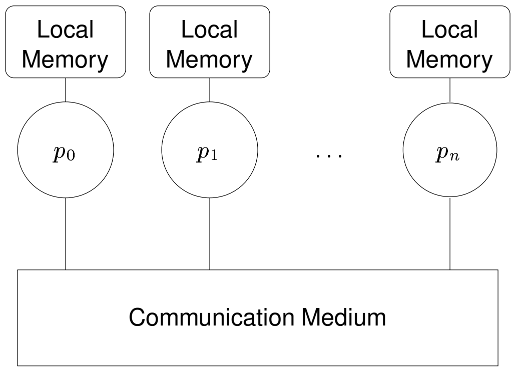

Figure 1.2: a message passing model

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 the 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 an 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 scenario 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. The longest chain of dependent events will hence give a measure for the 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 a link. However, it should be noted that in real systems it is not possible to guarantee an upper bound for messages traversing 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 scenario. The size of messages may also be interesting if it is not constant.

Causal relations are interesting in distributed computing due to the lack of a global state in an asynchronous system. They show which events for each process were caused by an event of another process e.g. in a message passing system: the event of receiving a message causes the process to do some local computation and/or send 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 be the same order in which the events really occurred, but it will lead to the same result of computation.

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 woken up. It distributes 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 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 a straight forward way of performing a broadcast. The initiator sends to all its neighbours a *broadcast* message. When a process receives *broadcast* message 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 an 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 *broadcast* messages to all its neighbours. Also processes which receive a *broadcast* message 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 of *broadcast* messages it sends 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, and the process sends an acknowledgement to its parent node and terminates the algorithm. Processes which receive a message *broadcast* despite having 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 the previous algorithm every process received the broadcasted information in time $O(D(l + d)$. Due to asynchrony the path on which the information is broadcast might be of length $O(n)$ as some messages might proceed on some links much quicker than messages on the shortest path (see example in figure 3.1).

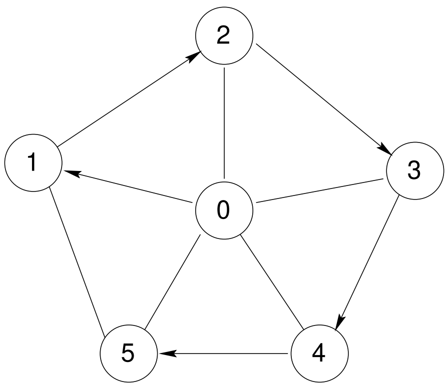

Figure 3.1: Example for a graph with diameter 2. Process 5 is guaranteed to receive a broadcasted message from process 0 in time $d + l$, but it is still possible that process 5 received the first *broadcast* message along the path indicated by the arrows.

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

With each process a unique identifier is associated in order to help the user 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 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 a 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 *thinking* state is not interested in its neighbours resources and sends requested forks to its neighbours. A process will switch from *thinking* to *hungry* state when it wants to access its resources. Then it tries to collect all forks of its neighbours. When a process in *hungry* state has collected all forks from its neighbours it will start accessing its resources. It changes to *eating* state in which it keeps all forks until it finishes with accessing the resources and changes to *thinking* state again.

The main problem is how to resolve starvation scenarios for two hungry processes competing for a resource. The following algorithms are solutions with different qualities concerning time complexity and fault tolerance. It should be noted that in the case described exactly two processes can share a resource, but the more general case where more than two processes 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 *request* message 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 occur in a different order than the received lexical order of events, it still results in an equivalent execution. Hence, on the receiving of a *request* message 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 *request* message from process q does the following depending on its state:

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

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

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

Correctness

The algorithm works correctly if it guarantees mutual exclusion, progress and starvation freedom. These properties are shown by the 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 send *request* messages to all their neighbours. In particular they send *request* messages to each other. Let t_p denote the local clock time when process p sends its *request* messages and let t_q denote the local clock time when process q sends its *request* messages. Due to the lexical order of time either $t_p < t_q$ or $t_p > t_q$ is valid.

If $t_p < t_q$ is valid then p would delay replying the *request* message of process q until p finished accessing its critical section, this implies that $t_p > t_q$ is valid. However, if $t_p > t_q$ is valid q would delay replying the *request* message of p until q finished with accessing its critical section, thus 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 where no messages are sent any longer and no process changes its state. Let p be a process in *hungry* state with the smallest request time. All neighbours in *thinking* state must have answered the *request* messages of p with sending *fork* messages. As p is the process that sent its *request* messages with the smallest time tuple, also all processes in *hungry* state will send *fork* messages 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 a *request* message makes the local clock of a process at least one unit greater than the time stamp of the *request* message. Therefore, a process which sends *request* messages to all its neighbours will loose at most one competition with each neighbour whilst it tries 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 a *fork* message from all neighbours which sent their *request* messages earlier. Receiving all *fork* messages 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 one 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 scenario until all other $n - 1$ processes have entered their critical section once and received all *fork* messages. 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 it 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 of the views which were proposed in chapter 2. For each view a description of its usage is given in the following:

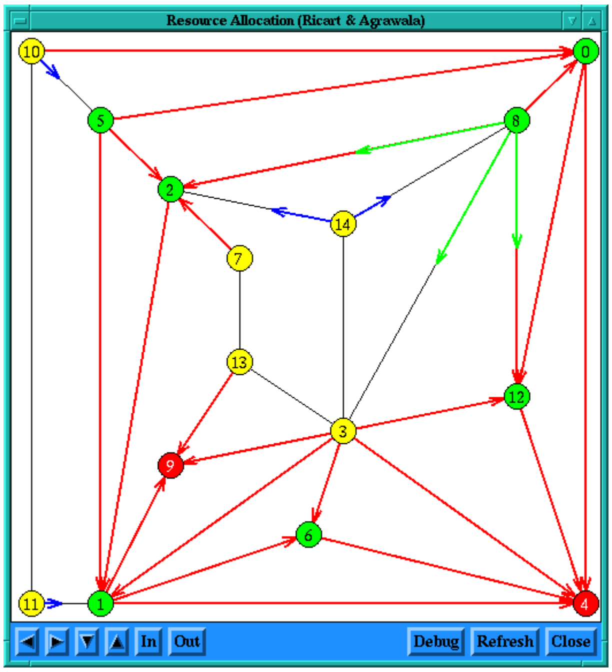

Figure 5.1: Basic view of the Ricart and Agrawala algorithm

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 become interested in a resource
- red when they have managed to access their critical section
- yellow when they have 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 by 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 flashes 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 *request* messages sent by all processes for accessing a resource once. A mark shows the maximum average value during the execution. It is 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 the form of arrows pointing from the sender of a message to the receiver. It starts at the local clock time of the sender when sending the message and ends at the 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 differently to 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 use the associated colour of the receiving process. This helps the user to see which relations were caused by which.

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 the 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 spent 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 the colour than the first part.

5.4 $\delta + 1$ Colouring by Luby

The $\delta+1$ colouring problem is concerned with 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 than the colour of its neighbour and the longest chain of processes that can 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 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 its neighbours which have not decided on a colour

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

The algorithm proceeds in phases and finishes when all processes have 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 $colour(p) := temp(p)$ and sends a *colour* message including $colour(p)$ to all its neighbours in $neighb(p)$.

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

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

Correctness

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

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

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

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

As q and p choose their values independently

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

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

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

□

From lemma 5.4 it can be concluded that in each phase $\geq \frac{1}{4}$ processes are expected to decide for a colour. Hence, all processes are expected to decide after $O(n \log n)$ phases. It remains to be shown 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 $colour(p) = colour(q) > 0$. 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 $colour(p)$ such that the q would have removed $colour(p)$ from $avail(q)$. Therefore, q could never have decided for a value which equals $colour(p)$. For the same reason q would not have determined a colour value earlier than p.

As $colour(p) = colour(q)$ 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 $colour(p) = colour(q) = 0$ which leads to a contradiction. □

Time Complexity

In each phase a process which has not determined a colour will send *colour* messages to its neighbours, receive *colour* messages from its neighbours, send *confirm* messages to its neighbours and receive *confirm* messages until it starts with the next phase or decides on 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 at the 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 a *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 such a way that the precedence graph remains 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 of neighbouring 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 reverses 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 its neighbour processes and wait until it has received all forks.

3. A process which has 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 a process's state is different from *eating* and f is dirty then f will be sent to the requesting process. If a process's state is also *hungry* then the request token will also be sent back.
- *Receiving a fork f:* The state of f will be changed to clean.

Correctness

The algorithm is correct if mutual exclusion and starvation freedom i.e. fairness, is guaranteed. The mutual exclusion property follows directly from the acyclic precedence graph, thus it remains to be shown 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 automatically 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 therefore 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 a predecessor. The proof will show, by induction, that a process of depth k will eventually eat if the 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 held by p. Hence p will keep it until entering the critical section.

2. The fork is in transit from its 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 a lower precedence a request token is replied to straight away by sending the fork. The only exeption is when q is eating. Then, process q delays sending the fork until it has left the critical section.

Therefore p receives at the latest after time $2(d + l)$, all forks, and manages to eat.

k-1 → k: Assume all processes at depth $k - 1$ manage to eat and process p is of depth k when it tries to access its resources. Each fork f which p tries to collect knows the states dirty or clean.

case f is clean: If f is held by p it remains by p until leaving the critical section. Otherwise, it is held by a neighbour q with higher precedence. Process q manages to eat and has to give p higher precedence to the fork. Hence, on a request the fork is sent to p and remains there until p can eat.

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

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

Time Complexity

From the correctness proof it can be concluded that a process at depth k has to wait for forks from its 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 the 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 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 stay 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 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 others 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 on a colour it is continuously flashing between a dark and a light blue. When a colour is determined the colour of a process changes to yellow again.

The colouring phase for a process p is finished when all neighbour colours are received. Hence p initializes 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 a red 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 held.

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

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

Similarly to the initialization for each edge a red solid or dotted arrow indicates which node is the holder of the fork. If a fork is in transit from one process to another this is

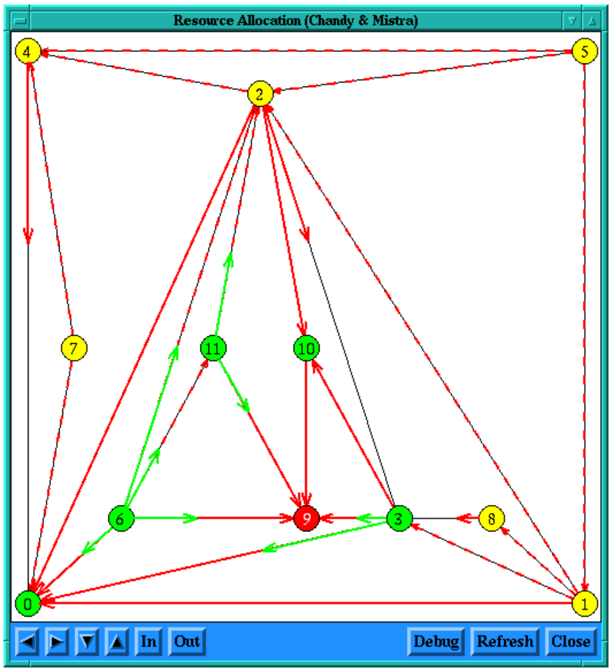

Figure 5.2: Basic view of the Chandy and Mistra algorithm

shown by a solid arrow which continuously resizes 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 recognizes this by observing a resizing arrow which is flashing between red and blue.

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 needs 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 sent 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 observes that the number of sent request tokens might strongly differ between two attempts at accessing resources. However, the shown marks 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 these events occurred. An arrow points 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 changes when a message is received. Then, the local clock is set to

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

The colouring of a relation depends on the event which occurred. 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 "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 *hungry* state and finishing with 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 is shown in the associated colour of a process when it tries to access resources. The second part uses the same colour in a lighter shade indicating that a process 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 $\delta+1$-colouring conflict solution should reduce the maximum access time. As described for the algorithm by Chandy and Mistra (see section 5.5) a $\delta + 1$-colouring results in an acyclic precedence graph. In contrast to Chandy and Mistra the precedences remains 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 guarantees that processes with lower precedence loose at most one competition with each higher neighbour and consequently do 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 pass a doorway at time t prevent neighbour processes entering the same area at a greater time than t until exiting the doorway. Thus, a doorway guarantees for a process that after a certain point in time no neighbours access an area. However, it is still possible that in spite of using a doorway neighbour processes occur in an area, since moving from one area to another takes some time.

Asynchronous Doorway

An asynchronous doorway is implemented, by requiring for each process p which desires to enter the doorway, a check on its 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 $q_0, \ldots, q_l$ exit the doorway. The code for entering and exiting this doorway is 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, $0 \leq i \leq l$, do not block p after leaving the doorway even if q_i tries straight after leaving to enter the same doorway again. In the worst case scenario q_i and p enter the doorway at the same time, since p considers only processes which passed the doorway before it starts trying. For the same reason any other neighbour process r cannot prevent p from passing the doorway when r passes the doorway, after p checked the state of its neighbours.

Attempting to use such a doorway together with a $\delta + 1$-colouring algorithm for solving the resource allocation problem leads to a solution where processes may have to wait a time which is exponentially in δ. The reason is given by the possibility that a process p may be blocked successively by lower coloured neighbour processes after p has 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 $T(c)$ denotes the maximum time which a process of colour c has to wait. Assume every process with colour > 0 has exactly $\delta - 1$ neighbour processes with a lower colour. Further, let us assume an execution where each process are successively blocked by its lower coloured processes. Then a process with colour c needs time

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

and therefore

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

until a process is able to access its critical section.

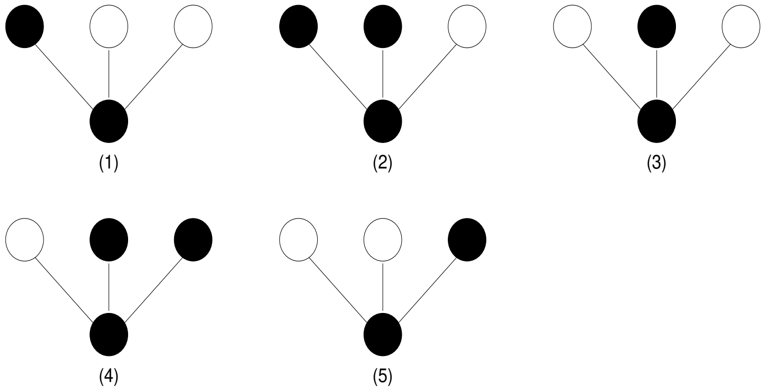

Figure 5.4: Example of a process being successively blocked in the doorway by its neighbours. The filled circles denote processes which have entered the doorway. Note that processes which enter the doorway in (2) and (4) must have been waiting for some other processes leaving the doorway.

Synchronous Doorway

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

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

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

Figure 5.5: Synchronous doorway entry code

Note that it is still possible that more than one process passes the doorway at the same time since all processes might find their neighbours outside the doorway and therefore could pass 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 use the graph structure of figure 5.6. It is possible that q and r cooperate such, that at least one of them is inside the doorway. Therefore process q waits until r is inside the doorway before exiting (this is possible since q and r are not neighbours and thus do not prevent each other passing the doorway). and r will do the same before it is exiting the doorway. Hence p is never able to gain excess to the doorway.

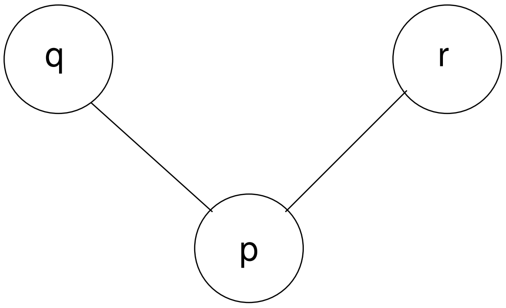

Figure 5.6: Example of a graph structure where two processes q, r can hinder process p from entering a synchronous doorway when either p or q will be inside the doorway.

Double Doorway

By using the previously described doorways for resource allocation either processes can 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 needs some further selection criteria for processes such as even doorways. The double doorway, a combination of the previous described doorways, prevents processes from being blocked for long periods inside or outside the doorway and hence leads to a solution of the resource allocation problem in which higher fault tolerance and faster access time to the critical section can be achieved.

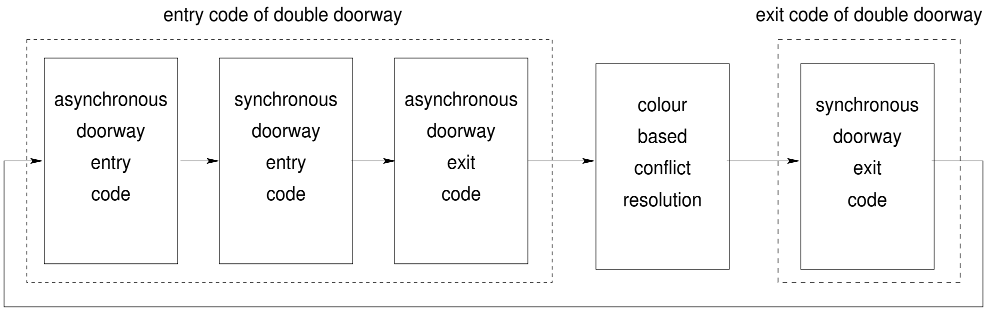

Figure 5.7: A double doorway together with a static colour based conflict solution.

As shown in figure 5.7 the entry code consists of the synchronous doorway entry code embedded in an asynchronous doorway entry and exit code. Processes which wish to enter the double doorway cannot forever be hindered from entering by neighbour processes, as the asynchronous doorway allows 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 is blocked from entering the synchronous doorway at most once by each neighbour. If a process q blocked 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 disallows q to pass until q has passed the asynchronous doorway exit code. Also, the problem of successive blocking after passing the doorway is 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 previously described double doorway one can gain directly an algorithm for solving the resource allocation problem. However, the double doorway can be optimized according to which neighbours are blocked at which kind of doorway. It is therefore, saving unnecessary blocking of processes and messages by the implementation. In the synchronous doorway one wishes to avoid the higher coloured processes starving because of lower coloured processes when using the static precedence graph. On the other hand the asynchronous doorway guarantees that lower coloured processes are not blocked forever by higher coloured processes passing the synchronous doorway. As a result the synchronous doorway blocks only lower coloured neighbours while the asynchronous doorway blocks only higher coloured neighbours.

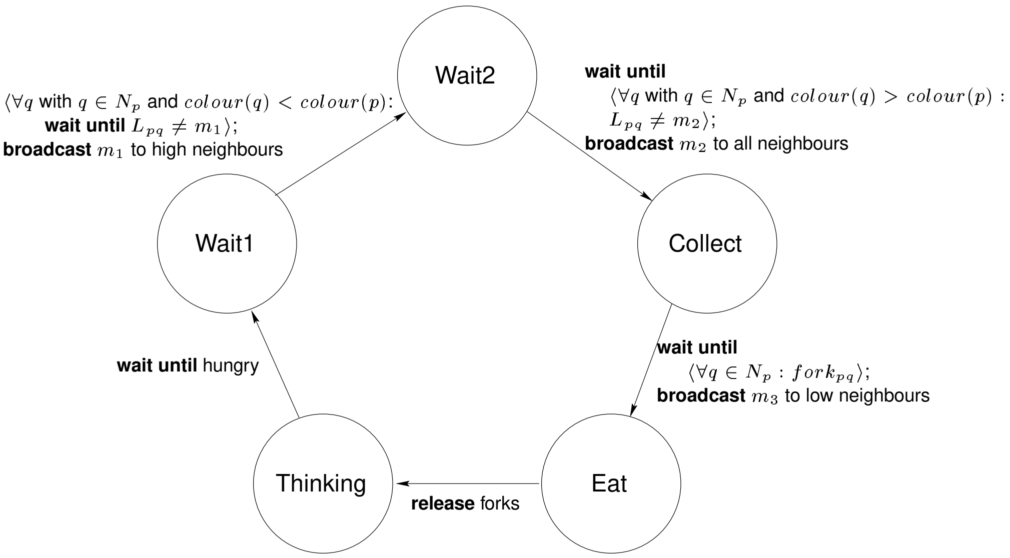

Figure 5.8: The state diagram of an algorithm with fault tolerance δ

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 *thinking* state 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 *eat* state 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 *wait1* state a process waits for its lower coloured neighbours inside the asynchronous doorway (in *wait2* state) to exit the asynchronous doorway (meaning neighbours change their state to *collect*).

- If a process is in *wait2* state a process waits until no higher coloured neighbours are inside the double doorway (no higher coloured neighbour is in *collect* state). Process p of *wait2* state 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 *collect* state process p tries to receive access to its resources. By sending m_2 messages to all its neighbours before entering *collect* state, p ensures that on the one hand all processes know about p requesting its forks, and on the other hand lower coloured processes are blocked from passing the synchronous doorway. When p is in *collect* state and has received all forks it signals to 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 updates its local state variable $L_{pq} := m_i$. Since message m_2 also equals a *request* message, p in *thinking*, *wait1* or *wait2* state would also send a *fork* message to q if $fork_{pq} = 1$. In *collect* state p only sends a *fork* message if $fork_{pq} = 1$ and $colour(p) > colour(q)$ holds. When p is in *eat* state it will send *fork* messages to neighbour $q \in N_p$ after leaving its critical section, if $fork_{pq} = 1$ and $L_{pq} = m_2$ are 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 is 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 enters its critical section only if it possesses all forks which are shared with neighbours, none of the neighbour processes enter the critical section at the same time. Also, a process inside its critical section keeps all forks until it has finished eating. Thus mutual exclusion is guaranteed. □

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

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

Proof. A process that has changed to *collect* state sends a m_2 message to all lower coloured neighbours in order to signal to them not to enter the *collect* state. Hence, a process has at most $(d + l)$ time units to change to *collect* state after a higher coloured neighbour sent m_2. Thus, process p of *collect* state 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 *collect* state, no other lower coloured neighbour in $\{q_0, \ldots, q_l\}$ reaches *collect* state. The longest of those paths decreases by one latest after time $2(d+l)$ since the lowest coloured processes directly manage to access their resources after requesting them and processes inside the critical section transmit forks after time $(d+l)$. By induction p needs 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 *wait1* state can be prevented from proceeding to *wait2* state only by lower coloured neighbours in *wait2* state. Further, a process in *wait2* state is only be hindered by processes in *collect* state if they are of higher colours. Processes hindering lower coloured neighbours in proceeding to *collect* state are not able to enter *wait2* state until the blocked lower coloured processes have managed to proceed to *collect* state due to the asynchronous doorway. Hence, if a process of *collect* state manages to eat, all other processes manage to eat and no process starves. This was the result of lemma 5.9. □

Time Complexity

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

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

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

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

□

Lemma 5.12 *A process in* wait1 *state manages to enter* wait2 *state 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 *wait1* state and $\{q_0, \dots, q_l\}$ denote the set of lower coloured neighbours known to be in *wait2* state when p became hungry. Process p is allowed to proceed to *wait2* state when $q_0, \dots, q_l$ managed to proceed to *collect* state. According to lemma 5.11 this happens 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 shows 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 scenario a process fails when it has collected all forks. This can happen in any state as a process releases only requested forks. Neighbour processes are not able to proceed further than *collect* state since they can never be able to collect all forks and thus never manage to eat.

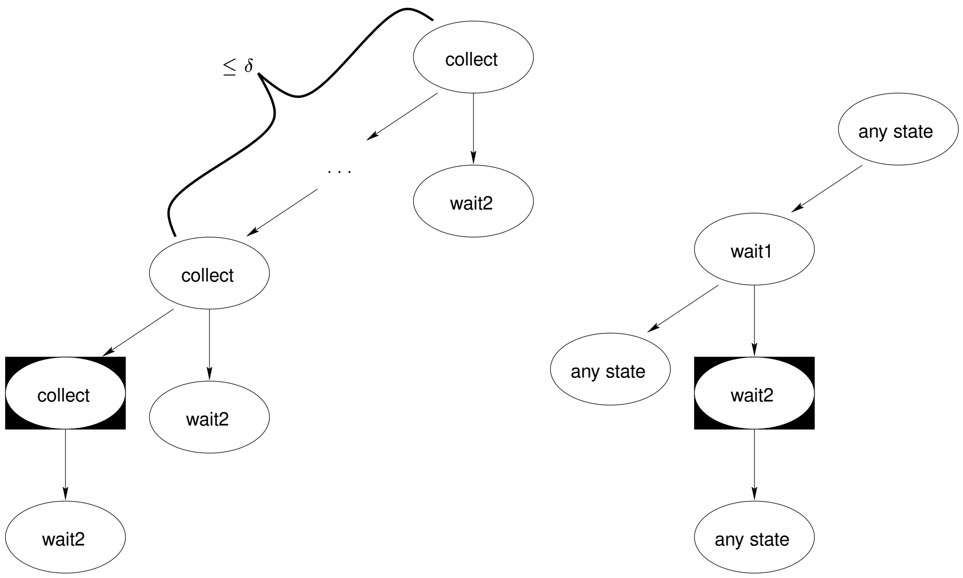

Figure 5.9: This diagram shows how the neighbourhood is affected when a process remains forever in *collect* or *wait2* state. The arrows show precedences between processes i.e. whether the neighbour process uses a higher or lower colour.

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

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

Using figure 5.9 it can be concluded that a process does not starve if none of its neighbours remain forever in *collect* or *wait2* state and neighbours do not fail. Further, a failing process affects other processes to remain forever in *collect* state only if they are not further than $\delta + 1$ because only higher coloured neighbours are affected in this sense and the maximum colour is limited by δ. Hence, a process affects processes to remain forever in *wait2* state only if they are not further than $\delta + 2$ since a processes remains in *wait2* state only if a neighbour process remains forever in *collect* state.

Therefore, a process manages to eat if no processes in the neighbourhood of $\delta + 4$ fail. 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 *collect* state could not eat. This allowed chains with δ processes in *collect* state 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 guarantees a better fault tolerance than the previous of 5.7.1, a process needs 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 proceeds 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 releases the fork.
2. If p receives 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 delays answering until
   - either p has 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 releases all requested forks and continues with 1.
4. If p receives all forks it eats and releases 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 is not to be interpreted as a *request* message and does not lead to releasing forks. For this purpose the algorithm sends extra *request* messages in *collect* state according to the previous described collection scheme. Assume process p receives a *request* message by q:

- if p is in *thinking*, *wait1* or *wait2* state it replies with sending a *fork* message and updating its value $fork_{pq} = 0$
- if p is in *collect* state and has lower precedence $(colour(p) > colour(q))$ it replies with sending a *fork* message and updating its value $fork_{pq} = 0$
- p also replies with sending a *fork* message and updating its value $fork_{pq} = 0$ if p has not collected all forks from lower coloured neighbours
- if p of *collect* state has higher precedence than q and has also collected all lower coloured forks it delays sending the *fork* message until it either manages to eat or it has to release a lower coloured fork before eating
- if p is eating it delays 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 the *collect* state is still guaranteed to enter its critical section in time $O(\delta(d + l))$. Considering that processes receive requested forks from higher coloured neighbours after time $2(d + l)$ the proof of lemma 5.9 explains why. Moreover, the new collection scheme does not affect processes in states other than *collect*. Therefore, it can be concluded from lemma 5.10 that the algorithm is still free from starvation.

Time Complexity

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

Message Complexity

A process has 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 has to request forks from all its neighbours. Due to the new collection scheme a process must release its forks whenever a lower coloured neighbour requests a fork. This could lead to an exponential size of messages, if the period until a process is 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 a link. Then a process is able to receive a requested fork from a neighbour in time $\geq 2(s+v)$. In the worst case scenario a process has to release forks straight after receiving them. A process only sends *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 has 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 remains constant.

Fault Tolerance

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

- q failed
- q is inside its critical section
- q is a 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 are not be able to collect all forks. A neighbour process p of a failing process which enters the *collect* state, is not able to proceed any further and thus remains forever in the *collect* state. Although p is never able to collect all forks, it can manage to collect at least all lower coloured forks (only if the failing process was of higher colour) and for this reason causes a higher coloured neighbour process q, to remain forever in *collect* state. Noting that q is never able to gain access to all its lower coloured forks (because of p), neighbours r_i of q behave in the following way:

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

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

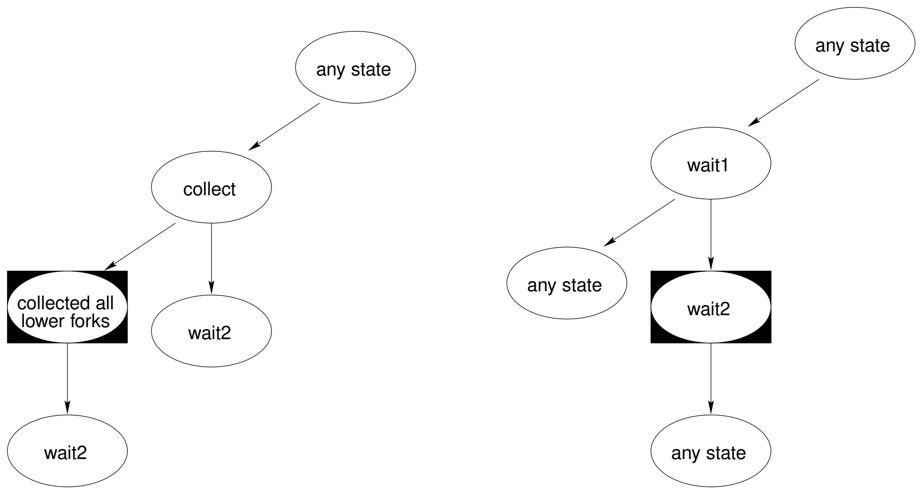

Figure 5.10: This diagram shows how the neighbourhood is affected when a process remains forever in *collect* or *wait2* state. The arrows show precedences between processes i.e. whether the neighbour process uses a higher or lower colour.

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 similarly and for this reason only one description for both algorithms 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 may still perform the colouring algorithm while others have already started with the main part.

In the following a description is given for each view. It should be noted that the basic view for these algorithms consists 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 the states of processes. These views are 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 the basic view usually does. Inside the main window processes 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 are 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 appears in colour yellow again. If neighbour processes determined a colour as well, the