Endochrony of Distributed Systems

DIPLOMA THESIS

Marlee Nathalie Basurto Macavilca

Tutor:
Prof. Dr. Klaus Schneider
M. Sc. Yu Bai

May 2014

(es) Embedded System group
Department of computer science
University of Kaiserslautern – Germany

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.
Kaiserslautern, den 26.05.2014

Marlee Basurto Macavilca

TABLE OF CONTENTS

LIST OF FIGURES:

1 INTRODUCTION

1.1 Motivation

Through the years the embedded system technologies have been developing and evolving, leading to very complex systems. Due to the complexity of the embedded system and hardware design, the designer split up the complex system into smaller systems. This is one of the reasons that we nowadays have a great number of distributed systems.

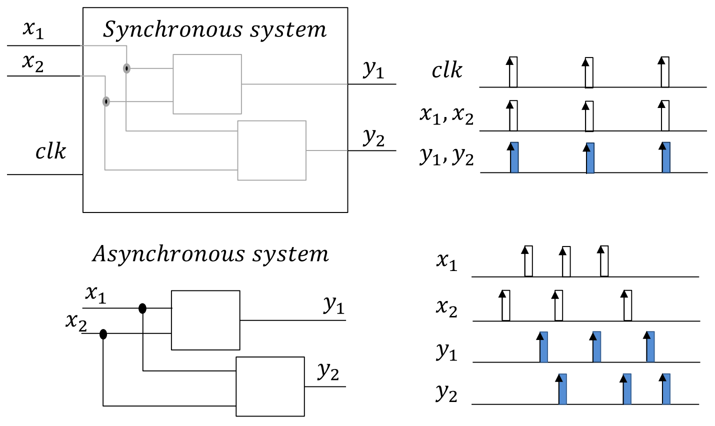
Therefore, instead of having a larger design which requires time consuming simulation and verification, the new design divides the complex system into simpler systems. The simpler systems are called components in order to differentiate from the complex system. The advantage of those systems is that as they can be placed in different locations, they can run independently from each other.

The other advantage of working in an asynchronous environment of the distributed system is that all the components do not have to wait for the $worst\ clock\ time$. To understand the term $worst\ clock\ time$, it is necessary to compare the two kinds of distributed systems in the network depending on the computation model: the synchronous system and the asynchronous system.

The synchronous system is controlled by a unique global clock and the global clock is constrained by the slowest component. All the others components of the system must wait to make the communication to an external environment as well as other components, until the slowest component ends its activity. After all the components have finished their activity, the whole system can interact at one point of time to the others components and to the external environment. On the other hand, we have the asynchronous system where every component runs independently without controlled by a global clock. Then, each component communicates with the environment or the other components at different points of time.

We see the Figure 1.1.1 where, it explains both models of computation. In the upper part there is a synchronous system and its behaviour. The behaviour of the synchronous system consists of three cycles. The system reads the input x_i and writes the output y_i when the global clock $clock$ ticks, it is every period cycle. In the lower part, there is an asynchronous system, where there is no notion of clock. The behaviour of the asynchronous system consists of three cycles too, but in different point of time as we see in the Figure 1.1.1.

Besides the complexity, the communication that used to play a second role in the technology plays now the dominant part. We want to have a more efficient communication in terms of speed and less power consumption, leading to a lower communication cost. The asynchronous systems fit those requirement explained.



Where:
↑ *Clock ticks*
↑ *Process of reading the input:* $x_i \wedge i \in \{1,2\}$
↑ *Process of writing the output:* $y_i \wedge i \in \{1,2\}$

Figure 1.1.1. The Synchronous and the Asynchronous modal of computation

However, we do not really want to remove or forget all the synchronous systems. We want to reuse some of them and adapt to the new design. The reason is the synchronous systems have been well researched for a long time and it is now well stated. Synchronous systems provide us with many tools to analyse and to verify the correctness of the implementation of the design. On the other hand, the distributed systems are difficult to design because they run asynchronously by nature. For those reasons, we want to use synchronous technique to implement the asynchronous distributed systems. Therefore, we reuse the synchronous system and make them work in an asynchronous distributed system.

To this end we verify if synchronous designs can be reused in an asynchronous environment without changing the origin behaviour. Is it easy to adapt synchronous systems into asynchronous systems? and have a correct *behaviour* of the system at the end of the conversion? Let us see the evolution of the synchronous system step by step.

From synchronous systems to the asynchronous systems:

We start from a synchronous system. The synchronous system reads all the inputs, updates the local variables and writes all the outputs at one point of time. It is called one $reaction$ because it the computation is does not consume any time. Therefore the computation in synchronous system is instantaneous. As we see the in the behaviour of the synchronous systems in Figure 1.1.1.

The second step is to divide the synchronous system into parts, called components. Each component is able to do an activity or computation and all of them work together to fulfil the task of the system. The computation of the whole system is done synchronously. The components work in parallel to each other. Each component has its own input and output and each of them have synchronous connections.

Now, it is time to convert the synchronous connection into FIFO (*first input- first output*) buffer communication. This process is the third step that it is called desynchronization of the synchronous system, see Figure 1.1.2.

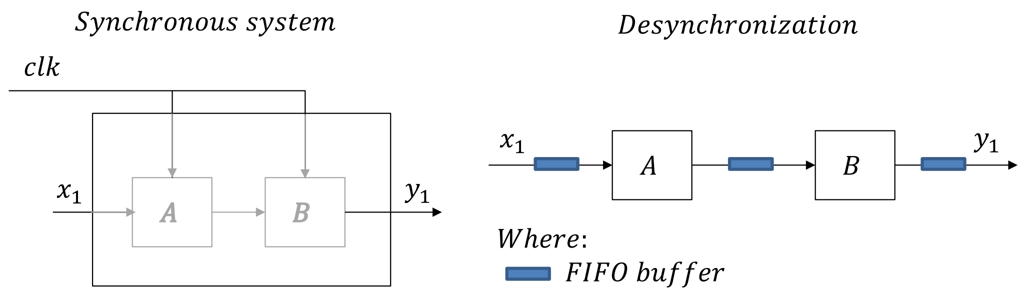


Figure 1.1.2. Desynchronization of Synchronous system

The desynchronization is the relation between the synchronous system and the asynchronous system. The synchronous connections are replaced by the FIFO buffer after making the desynchronization, as we in Figure 1.1.2 . Then, each component A and B has its own input and output and has its own clock. After the desynchronization, synchronous systems work in asynchronous environment.

Besides that the synchronous system works in an asynchronous environment, the desynchronization avoids processing irrelevant inputs values or to output irrelevant outputs values.
Let us explain the process of desynchronization following the $\boldsymbol{Sequential\ AND}$.

$\boldsymbol{Sequential\ AND}$ - $Synchronous\ model$:

$Firing\ rule$:

	x_1	x_2	y
α_1:	$(1 :: A)$	$(b\ ::\ B)$	$[b]$
α_2:	$(0 :: A)$	$(b\ ::\ B)$	$[0]$

$Flow\ of\ inputs\ and\ output$:

$\xi(x_1)$	1	0	0	1	…
$\xi(x_2)$	1	1	0	0	…
$\xi(y)$	1	0	0	0	…

The firing rule is a firing table and it is composed of rules α_i. The rule α_i fires one it fulfills n inputs conditions $x_i \in \{x_1, \dots, x_n\}$. Once the inputs conditions match, the system follows to write in the output $y_i \in \{y_1, \dots, y_n\}$ following the rules α_i fired.

$Sequential\ AND$ has two rules: $\alpha_1$and α_2, where $x_i \in \{x_1, x_2\}$ are the inputs; and $y_i \in \{y_1\}$ is the output. In order to fire the first rule α_1: the first input channel x_1 must be 1 and any value b in input x_2. After having both input values, it writes the value b in output y. The second fire α_2 fires when it has 0 in its first input channel x_1 and any value b in input channel x_2. After having both input values, it writes 0 value in output y. The firing rule explains the behaviour of the $Sequential\ AND$. As we can see on the right side of the table, the sequence of $reactions$ of the logical component AND. Each coloured column represents one $reaction$.

In one $reaction$, the synchronous system reads one value in all the inputs and writes as well one value all the outputs. Then, it reads in the first $reaction$ or round: $x_1 = 1, x_2 = 1 \overset{write}{\Longrightarrow} y = x_2 = 1$. Second $reaction$: $x_1 = 0, x_2 = 1 \overset{write}{\Longrightarrow} y = 0$. Third $reaction$: $x_1 = 0, x_2 = 0 \overset{write}{\Longrightarrow} y = 0$ and so on. As we can see neither in the second or third $reaction$, the system needs to read the input x_2 to output y. Therefore, the input x_2 is irrelevant to the computation when $x1 = 0$. Because of it (x_2) does not participate in the computation. It can be replaced by other value, and the system would behave as before.

$Synchronous\ System - introducing\ the\ \boxdot$:

We have seen that it is not necessary to read the input x_2 when the system reads 0 in x_1. If $x_1 = 0$ then the input x_2 becomes irrelevant for the computation. On the other hand, the system still needs the input x_2 for keeping the synchronization alive. For that reason, we replace the irrelevant value by introducing the symbol $\boxdot$.

x_1	x_2	y
$(1 :: A)$	$(b\ ::\ B)$	$[b]$
$(0 :: A)$	$(\boxdot :: B)$	$[0]$

$\xi(x_1)$	1	0	0	1	…
$\xi(x_2)$	1	$\boxdot$	$\boxdot$	0	…
$\xi(y)$	1	0	0	0	…

$Desynchronization - removing\ the\ \boxdot$:

Now, we want to remove the $\boxdot$ and work in an asynchronous model.

x_1	x_2	y
$(1 :: A)$	$(b\ ::\ B)$	$[b]$
$(0 :: A)$	B	$[0]$

$\xi(x_1)$	1	0	0	1	…
$\xi(x_2)$	1	0	…		
$\xi(y)$	1	0	0	0	…

The asynchronous model is not controlled by a global clock anymore. It means that each component of the network has its own clock, called a local clock. The local clock permits each component to run as fast as its own clock allows it. This model takes into account that the process of sending/receiving the message as well as the computation may take time. We can see that this model is more realistic. However, it is difficult to predict the behaviour of those systems due to the asynchronous concurrency.

After the desynchronization of a synchronous system, the processes should be able to cooperate and exchange messages independently with the other components and at the same time be able to behave as a unique synchronous single system. It is called the resynchronization or the $correct\ behaviour$ of the desynchronization.

The resynchronization reconstructs an asynchronous sequence of $reactions$ into a synchronous sequence of $reactions$. Informally speaking, endochrony is the criterion to decide whether a synchronous system is able to work in an asynchronous environment in a correct way or not. Intuitively, endochrony ensures that there is a unique way to resynchronize the input flows of the synchronous system in order to fire a unique output value in each reaction. As a result, the resynchronization has the same $behaviour$ as the original synchronous system. A sequence of reaction form a stream or flow and it is the $behaviour$ of the system.

Let us build the resynchronization of $Sequential\ AND (after\ the\ desychronization)$: One column does not necessarily mean one $reaction$. One $reaction$ after the desynchronization means: read only the input value x_i that permit to fire a specific rule α_i. The rest inputs are blocked until they are needed for the computation.

The system waits to read x_1 first. The first $reaction$: It reads in $x_1 = 1$ and waits to read the value in x_2, it is $x_2 = 1\ and \overset{write}{\Longrightarrow} y = x_2 = 1$. The second $reaction$: $x_1 = 0$ then, the system does not read the value in x_2 and outputs directly $y = 0$. The third $reaction$: $x_1 = 0$, therefore it outputs $y = 0$ and it does not consume the value in x_2. The input x_2 is locked to read again. The fourth $reaction$: $x_1 = 1$, then the input x_2 is necessary to read => $x_2 = 0 \overset{write}{\Longrightarrow} y = x_2 = 0$. The construction of the $Sequential\ AND$ is chowed in the Figure 1.1.3.

$Asynchronous\ model$

$\xi(x_1)$	1	0	0	1	…
$\xi(x_2)$	1	0	…		
$\xi(y)$	1	0	0	0	…

$Resynchronization$ $\Longrightarrow$

$Synchronous\ model$

$\xi(x_1)$	1	0	0	1	…
$\xi(x_2)$	1	⊡	⊡	0	…
$\xi(y)$	1	0	0	0	…

Figure 1.1.3. Resynchronization of the $Sequential\ AND$

As we have seen the system must know when an input it is necessary to read or not. This is why it is necessary to have an interface program that tells the system when to read an input value. The interface program is called $wrapper$. There is a $wrapper$ in each component and it decide to lock the reading of an input or to permit the reading.

The $Sequential\ AND$ fulfils the endochronous property because, it is able to model the same synchronous behaviour as the original one. Therefore, it is able to work in an asynchronous environment while having the same synchronous output flow. However, it is not always the case.

Let us see $\boldsymbol{Parallel\ ITE}$: The first rule $\alpha_1$does not need to read the input x_3 to fire the output y. The fire α_2 – input x_2 and the fire α_3 – input x_1 are same as fire α_1 – input x_3. In each rule, one input value is not needed to fire one specific rule. Therefore, they are removed in the firing rule of the desynchronization.

$Synchronous\ model -\ Introducing\ \boxdot$:

	x_1	x_2	x_3	y
α_1:	$(1 :: A)$	$(b\ ::\ B)$	$(\boxdot :: C)$	$[b]$
α_2:	$(0 :: A)$	$(\boxdot\ ::\ B)$	$(c :: C)$	$[c]$
α_3:	$(a :: A)$	$(b\ ::\ B)$	$(b :: C)$	$[b]$

$Desynchronization$:

	x_1	x_2	x_3	y
α_1:	$(1 :: A)$	$(b\ ::\ B)$	C	$[b]$
α_2:	$(0 :: A)$	B	$(c :: C)$	$[c]$
α_3:	A	$(b\ ::\ B)$	$(b :: C)$	$[b]$

The following step it to analysis the resynchronization. Let us see if there is a unique way to resynchronize the input flow after the desynchronization:

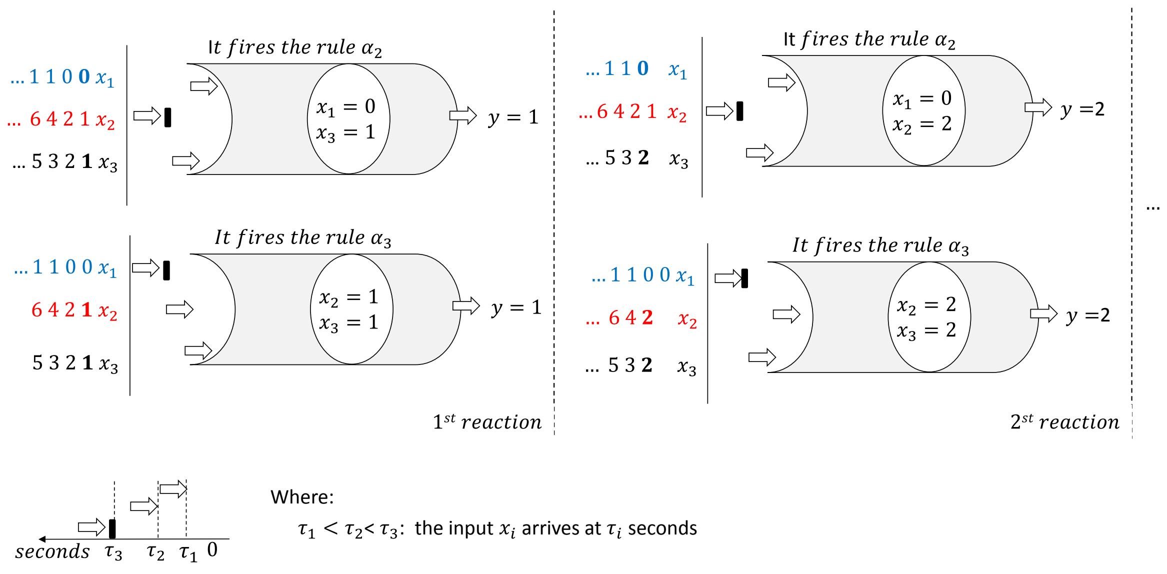
$\xi(x_1)$	0	0	1	1	1	0	…
$\xi(x_2)$	1	2	4	6	8	10	…
$\xi(x_3)$	1	2	3	5	7	9	…
$\xi(y)$	?	?	?	?	?	?	…

If the input x_1 is read first and is 1, then the system waits to read input x_2 to fire rule α_1. Otherwise x_1 is 0 and the system waits to read x_3 to fire rule α_2. The problem comes when the first read input is x_2 or x_3.

First $reaction$: The input x_1=0 is read first, then the system waits until read x_3 and fires the rule α_2. It output $y = x_3 = 1$. However, if the input x_3=1 is read first, there are two possibilities. The first possibility is: the input x_2=1 is read second and the system fires rule α_3. The second possibility is: the input x_1=1 is read second and the system fires rule α_2. The first and the second $reaction$ is show in Figure 1.1.4. As we see, it is a problem. Having the same input flow, the resynchronization system can fire different rules.

Depending on the arrival time of the input x_i, the flow of the rules is different. As a result we have a different output flows. The resynchronization is not successful. The $\boldsymbol{Parallel\ ITE}$ is not endochronous.

We show here the importance to analyse whether a synchronous system is endochronous or not. It is not trivial to remove the $\boxdot$ if the synchronous system is not endochronous. Finally, the aim of the thesis is to differentiate the distributed algorithms that are endochronous from those that are not.



Where:

$\tau_1 < \tau_2 < \tau_3$: the input x_i arrives at τ_i seconds

Figure 1.1.4. Two different resynchronizations from same input flow

1.2 Structure

The second chapter explains the theory of the two computation models of distributed systems: the synchronous model and the asynchronous model. It also explains the relation between the synchronous model and the asynchronous model of computation, called desynchronization. Finally the foundation chapter focus on the endochronous property.

The third chapter faces the desynchronization of many distributed algorithms. Those distributed algorithms have many applications in the real life, therefore it is important to analyse in detail whether they could desynchronize in a correct way or not. Most of them assume the communication and the behaviour of the process to be reliable while the rest take into account that there can be communication errors/mistakes. In each of them, we discuss whether it fulfils the endochronous property or not. At the end of this chapter, we will view in the all the distributed algorithms in order to determine which fulfil the endochronous property.

In case the distributed algorithms do not fulfil the endochronous property, the algorithms need an additional implementation. It is called the synchronizer, explained in chapter four. The synchronizer is able to make the synchronous design work in an asynchronous environment.

Finally, we summarize the entire thesis in chapter 5.

2 FOUNDATION

Distributed algorithms are algorithms designed to run on hardware consisting of many interconnected processors. There are two kinds of distributed systems that depend on the model of computation: the Synchronous model and the Asynchronous model. See in more detail in [2, 12]. The synchronous model, the processors are completely synchronous, performing communication and computation in perfect lock-step synchrony. On the other hand, the asynchronous system works completely asynchronously, taking steps at arbitrary speeds and in an arbitrary order.

Whereas we are working in a synchronous model or asynchronous model there is a notion of rounds /steps. The notion of round or step is defined as the action of reading the inputs, updating to the actual state and the local variables, and writing the output every certain time. In the case of synchronous system, it occurs always every certain time and all of its components do their computation at the same time. In contrast, all the components that belong to the asynchronous system has different clocks, therefore the action of reading, updating and writing occurs at different periods of time. Due to that, each of them has a round or step differently.

2.1 Synchronous model

Before explain in detail this model, let us show some examples of synchronous model in the nature.

- Synchronous of menstrual periods of group of women [8]
- Synchronization of heart pace-maker cells [11]
- Flashing of fireflies and shirping of cicadia [10]
- Self-organization of hand-clapping [9]
- Synchronization of metronomes [5]

Those examples show us that the nature tends to follow a synchronous model without introducing any external help.

On the other hand, the technologies have been trying to pursuit the synchrony and integrity in each devise, in order to have under control each component before communicating to an external device. Due to the long-time research, the synchronous system is a well know model.

It simplifies programming, since developers do not have to take care about low-level detail like timing, synchronization and scheduling. However, it has some consequences that make the compilation of synchronous program not at all straightforward. All the signals of the program should have a well-defined temporal behaviour, the *clock consistency*. Another important issue is the *causality analysis*. We assume that both characteristics are fulfilled by the problems in this thesis.

Coming back to the model of computation, the principal characteristic of the synchronous model is the notion of a global clock and the synchronous concurrency. The communication and the computation of the system are carried out synchronously, in each round, controlled by a unique global clock.

Let us see how the synchronous computations model works. Equation system:

$$\begin{cases} q^{(0)} := f(x^{(0)}, y^{(0)}) \\ q^{(t+1)} := f(x^{(0)}, q^{(t)}) \\ y^{(t)} := g\left(x^{(t)}, q^{(t)}\right) \end{cases}$$

Where:

Input $\mathcal{V}_{in} = x = (x_1, \dots, x_m)$
Output $\mathcal{V}_{out} = y = (y_1, \dots, y_n)$
State variable $q = (q_1, \dots, q_k)$

Besides, It is defined as $\mathcal{V} = \mathcal{V}_{in} \cup \mathcal{V}_{loc} \cup \mathcal{V}_{out}$ (input, local and output variable). The input $\mathcal{V}_{in} = \{x_1, \dots, x_n\}$, the output $\mathcal{V}_{out} = \{y_1, \dots, y_m\}$ and the local variable $\mathcal{V}_{loc} = \{z_1, \dots, z_k\}$.

Explanation of the syntax:

Before starting the first round, the system establishes the actual state according to the type of variable the default value $q^{(0)}$. In each period of time t, the system update the actual state $q^{(t+1)}$ based on previous state and default input value and write on the output $y^{(t)}$ is a function of the actual input and the actual state. In each cycle or each reaction the system must read one input value and write one output value, because the synchronous system is deterministic.

The equation systems essentially correspond to hardware circuits. And the synchronous circuit use the synchronous model of computation (MoC). Therefore, it is possible to generate efficient software and hardware from the same synchronous program. One clear example is hardware description languages like VHDL or Verilog.

The following step is to make clear about the system and the components:

$$P_{\|} = P_1 \parallel \cdots \parallel P_n$$

Where:

Synchronous system is $P_{\|}$

Component of the synchronous system $P_i \in \{P_1, P_2, \ \dots P_n\}$

The synchronous system $P_{\|}$ is composed by n components and all of them work in parallel.

Another characteristic of the synchrony circuit design is that every component of the system is controlled by a global clock. Moreover, the global clock is determined by the worst case execution time of all the components. The clock of all entire system is imposed by the slowest components; therefore the rest of the components are forced to wait until the slowest one finishes its computation. It is designated the component P_i as the slowest component because it needs more time than the rest to finish its computation.
Because, it has to permit all the signals propagate through the circuit before the next clock arrives and all of them must be ready to communicate. It is explained in more detail in [14].

Synchronous connection

In order to work with communication between components, we pay attention to the synchronous connection between every component in the system. We have seen that the synchronous system has its inputs and output, however if we see inside of the synchronous system we find its component P_i has its own inputs and outputs.

We notice that the component work in parallel with the other. The computation and the communication is done synchronously. Whenever the global clock ticks all the components make the instantaneous communication and computation in zero time.

Example of Synchronous modal of computation:

Let´s see the following example, **Sequential ITE (if then else)**

$Firing\ rule$:

x_1	x_2	x_3	y
$(1 :: A)$	$(b\ ::\ B)$	$(c :: C)$	$[b]$
$(0 :: A)$	$(b\ ::\ B)$	$(c :: C)$	$[c]$

$Flow\ of\ inputs\ and\ output$:

$\xi(x_1)$	1	0	1	1	…
$\xi(x_2)$	1	3	3	5	…
$\xi(x_3)$	0	2	2	6	…
$\xi(y)$	1	2	3	5	…

It has 3 inputs (x_1, x_2, x_3) and 1 output (y), each of them are indexed by a global clock. The input x is a Boolean value and the others, (x_2, x_3) are natural values.
In each cycle we consume one value of each input stream and generate one output value.

If $x_1 = 1 => y = x_2$, otherwise $x_1 = 0 => y = x_3$.

Let us see the computation in detail: We consume three values from the input but we use only two of them. The first cycle, we eat $x_1 = 1, x_2 = 1, x_3 = 0$, we know if $x_1 = 1 => y = x_2 = 1$ and we do not use the input x_3, however it is consumed. For the second cycle is the same, we do not to have to read the imput x_2 and so on. We realize that we are reading some values that are not interesting for the computation. Those values are called irrelevant values that does not affect to the flow output stream of the system. On the other hand, we have to evaluate if they are crucial for the synchronization of the system. We evaluate it in the section of desynchronization.

To this aim, we desynchronize the synchronous system and analyse its behaviour. The behaviour is formally defined as evaluate the input and fire the rule in each step. Each output value in considered as one step more or one more cycle.

The main conclusion to be drawn from the synchronous system is that it is well developed and it allows us to program, compile and verified the correctness of the implementation. Due to that, we try to keep the synchronous system and try to reuse it in a more efficient design, i.e. asynchronous environment.

2.1.1 QUARTZ [5]

There are various types of synchronous programs like ESTEREL, LUSTRE, SIGNAL and QUARTZ. ESTEREL can make an implementation and verification of reactive real-time systems, however it cannot make some modern verification methods as e.g. abstraction from certain data types that it yields in nondeterministic systems.

In order to solve it, the Group of Embedded System have developed a new “synchronous” language called QUARTZ that is very similar to ESTEREL [5]. In particular, QUARTZ added statements for asynchronous parallel execution of threads, and for explicitly implementing non determinism.

There are also some differences in the semantics of the data values that are used in QUARTZ and ESTEREL: it is the inmediate assignment and the delayed assignment, it is explained later.

In chapter 3, we use some the semantic and syntax of QUARTZ datatype:

Storage	
mem	Memorized variable (store last value
event	Event variable (store last values)
Information flow	
?	Input variable (only readable)
!	Output variable (only writable)
	Inout variable (readable and writable)
Data types	
bool	Booleans
nat	Unbounded unsigned integers
int	Unbounded signed integers

It is fully explained in [3]

2.2 Asynchronous system [5]

In contrast to the synchronous model, in the asynchronous model each component of the system has its own clock. In other words they can perform the communication and computation controlled by its own clock. There is still notion of round, however, each of them have its own counter round.

The component P_i is not controlled by the worst-case execution time component as the synchronous model, therefore P_i is not forced to wait to the lowest component.

Define as asynchronous environment:

$$P_{\parallel}^{a} = P_1 \parallel_a \dots \parallel_a P_n$$

Where:

Asynchronous system is $P_{\parallel}^{a}$

Component of the asynchronous system $P_i \in \{P_1, P_2, \ \dots P_n\}$

Each component $P_i$runs independently and without a specific order, but all together cooperates to achieve the specific objective or task of the asynchronous system.

It seems natural to have an asynchronous system but it is really complex to evaluate the correctness of the implementation due to the asynchronous concurrency. Then, we want to take full advantages of the synchronous systems. First, we reuse the synchronous system and adapt them in the new interface i.e. the asynchronous environment. With this intention, we desynchronize the synchronous system.

Generating DPN (Dataflow Process Network)

In order to work with communication between components, we generate DPN of synchronous systems. The Dataflow process network (DPN) is a model of computation where a number of concurrent processes communicate through unidirectional FIFO channels [7].

First, we view the system as a `hardware circuit´. Then, we construct a DPN of the `circuit´ by:

- Considering each component P_i as a single node of the DPN
- Replace the connection between the components by FIFO buffers
- When there is a fork on the connection, it must be implemented with Duplication nodes

Now, we see that it is essentially an asynchronous hardware, if we replace the connections as FIFO buffers.

See the example: $\boldsymbol{Sequential\ ITE} - Asynchronous\ system$

$Firing\ rule$:

	x_1	x_2	x_3	y
α_1:	$(1 :: A)$	$(b :: B)$	C	$[b]$
α_2:	$(0 :: A)$	B	$(c :: C)$	$[c]$

$Flow\ of\ inputs\ and\ output$:

$\xi(x_1)$	1	0	1	1	…
$\xi(x_2)$	1	3	5	…	
$\xi(x_3)$	2	…			
$\xi(y)$	1	2	3	5	…

The system $Sequential\ ITE$ must necessarily wait to read first the input x_1 . If x_1 is 1, it waits to read x_2 and fires rule α_1. Then, it outputs $y = b$. If x_1 is 0, it waits to read x_3 and it fires rule α_2. Then, it outputs $y = c$. As we see, we avoid to read one irrelevant input value in each rule α_i. The sequence of computations (stream) occurs by FIFO. Each read value is consumed value too.

2.3 Desynchronization

The desynchronization is the link between the synchronous system and the asynchronous system [13]. So after having found the irrelevant value in the computation, the next step is removed them. Then, the objective consists on removing the irrelevant value ($\boxdot$) and the synchronization boundaries of the reactions. Let us see better in the following example.

$\boldsymbol{Sequential\ ITE}$ - $Synchronous\ system$:

$Firing\ rule$:

x_1	x_2	x_3	y
$(1 :: A)$	$(b :: B)$	$(c :: C)$	$[b]$
$(0 :: A)$	$(b :: B)$	$(c :: C)$	$[c]$

$Flow\ of\ inputs\ and\ output$:

$\xi(x_1)$	1	0	1	1	…
$\xi(x_2)$	1	3	3	5	…
$\xi(x_3)$	0	2	2	6	…
$\xi(y)$	1	2	3	5	…

Because of being a synchronous model, all the inputs must read one value and write one value in the output in each round. The round of system is represented by one entire column of the $flow\ of\ inputs\ and\ output$.

It is time to recognize the irrelevant value and replace then by $\boxdot$.

The meaning of $\boxdot$:

- We still working synchronously. We label $\boxdot$ the irrelevant values that are not used for computation.

$\boldsymbol{Sequential\ ITE}\ -\ Synchronous\ system -\ introducing\ \boxdot$:

$Firing\ rule$:

x_1	x_2	x_3	y
$(1 :: A)$	$(b :: B)$	$(\boxdot :: C)$	$[b]$
$(0 :: A)$	$(\boxdot :: B)$	$(c :: C)$	$[c]$

$Flow\ of\ inputs\ and\ output$:

$\xi(x_1)$	1	0	1	1	...
$\xi(x_2)$	1	$\boxdot$	3	5	...
$\xi(x_3)$	$\boxdot$	2	$\boxdot$	$\boxdot$	...
$\xi(y)$	1	2	3	5	...

α_1 (the first firing rule): the value x_3 is needed for the synchronization, but not computation => then it are replaced by $\boxdot$. For α_2, x_2 is replaced by $\boxdot$.

The next step is to remove the irrelevant values $\boxdot$.

$\boldsymbol{Sequential\ ITE}\ -\ Desynchronization$:

$Firing\ rule$:

x_1	x_2	x_3	y
$(1 :: A)$	$(b :: B)$	C	$[b]$
$(0 :: A)$	B	$(c :: C)$	$[c]$

$Flow\ of\ inputs\ and\ output$:

$\xi(x_1)$	1	0	1	1	...
$\xi(x_2)$	1	3	5	...	
$\xi(x_3)$	2	...			
$\xi(y)$	1	2	3	5	...

After the desynchronization (removing the irrelevant value $\boxdot$), the system is able to work in the new interfade i.e. asynchronous enviroment. In fact, each reaction depends only on the values needed for the computation. It means, in each reaction they just consume the relevant values.

We see the stream desynchronized and they are still working properly, because the output stream is the same as the output stream of the synchronous version. It let us state that the node can resynchronize the stream. Therefore, it is endochronous.

On the other hand, there are some algorithms that could not transform into asynchronous system. They cannot resynchronize after the desynchronization.

2.4 Endochronous [4, 15]

Until now, it has been explained the process of desynchronization. However, we need to know whether it is a *correct* desynchronization or not. We have to see if it is possible to reconstruct a unique synchronous behaviour after the desynchronization. Besides that, we see if the communication behaviour between synchrony and asynchrony are equivalent. For this reason, we see the endochronous property. Informally speaking, endochrony is the property to resynchronize the asynchronized inputs deterministically.

Important definitions:

$Stream$: It is a flow data values. It can be an input data, local data or output data.

$Behaviour$: In order to define the behaviour of the system or component, we need streams. The stream of inputs data, local data and output data and the special value $\boxdot$. The symbol $\boxdot$ is used in synchronous model to replace the irrelevant value.

In order to formalize the definition, there are some notations. Consider behaviour ρ, ξ that map variables to streams:

Clock equivalence: $\rho \approx_{cl} \xi$, it is clock equivalence if after removing all the irrelevant values ($\boxdot$) from each stream at a point of time, they became to have the same streams.

Flow equivalence: $\rho \approx_{fl} \xi$, it is flow equivalence if after removing the irrelevant values from a particular stream $\rho(x) \vee \rho(y)$ they became to have the same stream.

The $\rho \approx_{fl}^{in} \xi$ means that ρ and ξ were obtaining by inserting $\boxdot$ arbitrarily in the input stream from the same stream.

Definition 1 (Endochrony)

A synchronous system P is called endochronous if for all $\rho_1, \rho_2 \in Bhv(P)$ with $\rho_1 \approx_{fl}^{in} \rho_2$ we also have $\rho_1 \approx_{cl} \rho_2$. In other words, the input flow equivalence implies clock equivalence.

Example $\boldsymbol{Sequential\ OR}$:

$Synchronous\ version$

x_1	x_2	y
$(1 :: A)$	$(b\ ::\ B)$	$[1]$
$(0 :: A)$	$(b\ ::\ B)$	$[b]$

$Introducing\ \boxdot$

x_1	x_2	y
$(1 :: A)$	$(\boxdot :: B)$	$[1]$
$(0 :: A)$	$(\ b :: B$	$[b]$

$Desynchronization$

x_1	x_2	y
$(1 :: A)$	B	$[1]$
$(0 :: A)$	$(b\ ::\ B)$	$[b]$

The last table show the result of the desynchronization of the synchronous system $Sequential\ ITE$. Now, it is time to reconstruct the input streams into synchronous reactions – it is called resychronization. The resynchronization of $Sequential\ ITE$ is able reconstruct a synchronous behaviour as we see in Table 2.1.

The synchronous streams after the deynchronization and resynchronization, respectively:

$\xi_1(x_1)$	0	0	1	1	1	0	…
$\xi_1(x_2)$	1	0	1	…			
$\xi_1(y)$	1	0	1	1	1	1	…

$\xi_2(x_1)$	0	0	1	1	1	0	…
$\xi_2(x_2)$	1	0	$\boxdot$	$\boxdot$	$\boxdot$	1	…
$\xi_2(y)$	1	0	1	1	1	1	…

Table 2.1. Behaviour after the desynchronization and resynchronization of $\boldsymbol{Sequential\ OR}$

We clearly see that $\xi_1 \approx_{fl}^{in} \xi_2$ and it fits also $\xi_1 \approx_{cl} \xi_2$, therefore it is endochronous .

Let us see one other example: $\boldsymbol{Parallel\ ITE}$

$Synchronous\ version$:

	x_1	x_2	x_3	y
α_1:	$(1 :: A)$	$(b :: B)$	$(c :: C)$	$[b]$
α_2:	$(0 :: A)$	$(b :: B)$	$(c :: C)$	$[c]$
α_3:	$(a :: A)$	$(b :: B)$	$(b :: C)$	$[b]$

$Introducing\ \boxdot$:

	x_1	x_2	x_3	y
α_1:	$(1 :: A)$	$(b :: B)$	$(\boxdot :: C)$	$[b]$
α_2:	$(0 :: A)$	$(\boxdot :: B)$	$(c :: C)$	$[c]$
α_3:	$(a :: A)$	$(b :: B)$	$(b :: C)$	$[b]$

$Desynchronization - Asynchronous\ version$:

	x_1	x_2	x_3	y
α_1:	$(1 :: A)$	$(b :: B)$	C	$[b]$
α_2:	$(0 :: A)$	B	$(c :: C)$	$[c]$
α_3:	A	$(b :: B)$	$(b :: C)$	$[b]$

Given the inputs $\xi(x_i)$ we could produce the three different stream outputs $\xi(y)$:

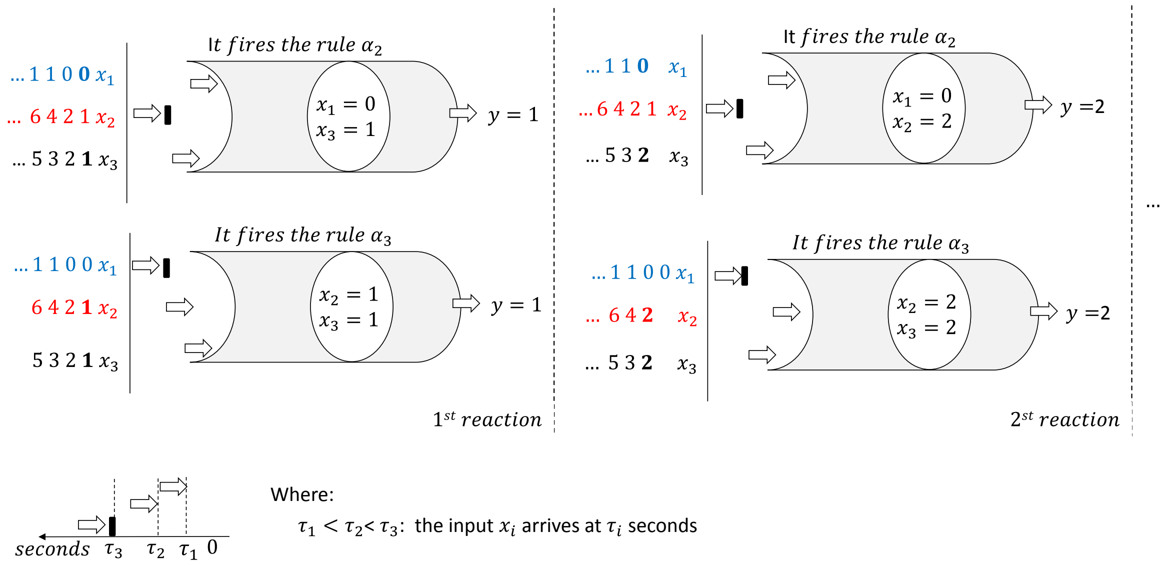
$\xi(x_1)$	0	0	1	1	1	0	…
$\xi(x_2)$	1	2	4	6	8	10	…
$\xi(x_3)$	1	2	3	5	7	9	…
$\xi(y)$	?	?	?	?	?	?	…

It is assumed to have this flow of inputs after the desynchronization. Let us see if there is a unique way to resynchronize to synchronous flow of outputs.

Let see in detail the first reactions. They are controlled by the firing rule of the Desynchronization:

	x_1	x_2	x_3	y
α_1:	$(1 :: A)$	$(b :: B)$	C	$[b]$
α_2:	$(0 :: A)$	B	$(c :: C)$	$[c]$
α_3:	A	$(b :: B)$	$(b :: C)$	$[b]$

We show in the following figure that: Depending on the arrival time of the input, it fires a different firing rule. The system does not have a unique way to resynchronize the input flow. In the figure, we explain the arrival time by τ_i . The consumed values are inside of the pipe and the other is locked for reading. We represent just two consecutive reactions from the same input flow.



Where:

$\tau_1 < \tau_2 < \tau_3$: the input x_i arrives at τ_i seconds

Figure 2.4.1. Parallel ITE: 2 different resynchronizations from the same input flow

We have showed that after the resynchronization, it has 3 different synchronous behaviours. See the following:

First behaviour:

$\xi_1(x_1)$	0	0	1	1	1	0	...
$\xi_1(x_2)$	$\boxdot$	$\boxdot$	1	2	4	$\boxdot$	...
$\xi_1(x_3)$	1	2	$\boxdot$	$\boxdot$	$\boxdot$	3	...
$\xi_1(y)$	1	2	1	2	4	3	...
Firing rule fired	α_2	α_2	α_1	α_1	α_1	α_2	

Second behaviour:

$\xi_2(x_1)$	$\boxdot$	0	0	1	1	1	...
$\xi_2(x_2)$	1	$\boxdot$	$\boxdot$	2	4	6	...
$\xi_2(x_3)$	1	2	3	$\boxdot$	$\boxdot$	$\boxdot$	...
$\xi_2(y)$	1	2	3	2	4	6	...
Firing rule fired	α_3	α_2	α_2	α_1	α_1	α_1	

Third behaviour:

$\xi_3(x_1)$	$\boxdot$	$\boxdot$	0	0	1	1	...
$\xi_3(x_2)$	1	2	$\boxdot$	$\boxdot$	4	6	...
$\xi_3(x_3)$	1	2	3	5	$\boxdot$	$\boxdot$	...
$\xi_3(y)$	1	2	3	5	4	6	...
Firing rule fired	α_3	α_3	α_2	α_2	α_1	α_1	

From the same inputs streams we have different outputs streams. There are three behaviour:

$\xi_1(y)$	1	2	1	2	4	3
Firing rule fired	α_2	α_2	α_1	α_1	α_1	α_2

$\xi_2(y)$	1	2	3	2	4	6	3
Firing rule fired	α_3	α_2	α_2	α_1	α_1	α_1	α_2

$\xi_3(y)$	1	2	3	5	4	6
Firing rule fired	α_3	α_3	α_2	α_2	α_1	α_1

Therefore the system $Parallel\ ITE$ could not divide into smaller component. It does not fit the endochrony. Besides, there is another option to check whether it is endochronous or not. It is by checking the firing rules.

The second way to ensure the correctness of the desynchronization is the following.

Endochronous by Firing rules:

A synchronous node is endochronous if and only if its desynchronized version does not have overlapping firing rules. It means, they have a unique way to fire or to react after reading the input or imputs.

	x_1	x_2	x_3	y
α_1:	$(1 :: A)$	$(b :: B)$	C	$[b]$
α_2:	$(0 :: A)$	B	$(c :: C)$	$[c]$
α_3:	A	$(b :: B)$	$(b :: C)$	$[b]$

The firing rule α_1 and α_3 are overlapping to each other, because it depends on the arrival time of inputs. If x_1 arrives first, then second x_2 and x_3, and $x_2 = x_3$, the system is confuse to whether fire α_1 => consume the inputs x_1 and x_2 and output $y = x_2$, or fire α_3 => consume the inputs x_2 and x_3 and output $y = x_2 = x_3$.
The same situation happens with α_2 and α_3, they are overlapping rules too. As a result Parallel ITE is not endochronous.

Endochronous modules in Asynchronous systems

The endochronous systems wait for a uniquely defined next input and can then determine a synchronous reaction. We can take into advantage of this characteristic to construct an endochronous wrapper $\mathcal{C}_P$ of the synchronous module P.

The wrapper:

- $\mathcal{C}_P$ observes the arriving inputs of P
- $\mathcal{C}_P$ triggers P when enough input values arrived
- $\mathcal{C}_P$ inserts for irrelevant values that have not been sent

Afterward, we can say that endochronous module P can be used in an asynchronous setting, it means that it can be used in asynchronous networks.

Endochronous wrappers are also called clock generators.

Examples of endochronous wrappers for **Sequential ITE**:

x_1	x_2	x_3	y
$(1 :: A)$	$(b :: B)$	C	$[b]$
$(0 :: A)$	B	$(c :: C)$	$[c]$

The wrapper waits until value x_1 arrives, second:
If $x_1 = 1$, then it waits the value b at input port x_2, as soon as b comes the wrapper sends $(x_1, x_2, x_3) = (1, b, \boxdot)$ to the local synchronous module and trigger it.
Otherwise, $x_1 = 0$ and the wrapper waits until arrives c at input port x_3 and send $(x_1, x_2, x_3) = (0, \boxdot, c)$ to the local synchronous module and trigger it.

Afterwards, the endochronous module can be triggered by its endochronous wrappers in an asynchronous environment. A further step is to check if the endochrony is compositional or not.

Endochrony is not compositional?

Consider a synchronous system $P_\parallel = P_1 \parallel \cdots \parallel P_n$ where the module P_i is endochronous. The following step is to check whether it is or not equivalent to $P_\parallel^a = \mathcal{C}_1(P_1) \parallel_a \dots \parallel_a \mathcal{C}_n(P_n)$, the asynchronous compositions of the endochronous components with their wrappers.

Example of $Copy1$:

x_1	y_1
$(a :: A)$	$[a]$

Wait to read the value a and write it. It is endochronous as there is no other firing rule to react.

Now, if $Copy = copy1 \parallel copy2$. $Copy1 = Copy2$ and they run in parallel.

x_1	x_2	y_1	y_2
$(a :: A)$	$(b :: B)$	$[a]$	$[b]$

$Copy1$ and $copy2$ are endochronous, then each of them can run in an asynchronous environment. However, we want to check if all the system together is endochronous.

$Copy_{\|}^{a} = \mathcal{C}_1(copy1) \parallel_a \mathcal{C}_2(copy2\,)$ in an asynchronous network. See in detail it:

	x_1	x_2	y_1	y_2
α_1:	A	$(b :: B)$	$[]$	$[b]$
α_2:	$(a :: A)$	B	$[a]$	$[]$
α_3:	$(a :: A)$	$(b :: B)$	$[a]$	$[b]$

α_1: it considers *copy1* has a different clock from *copy2,* and copy2 is faster than copy1.
α_2: The clock of *copy2* is slower than *copy1*
α_3: The clock of *copy2* and c*opy1* is the same

There are two forms to prove the system *copy* in not endochronous.

- The firing rules: α_1 and α_3 are overlapping to each other and α_2 and α_3 as well.
- By checking the forma definition of endochrony: $\xi_1 \approx_{fl}^{in} \xi_2$ implies $\xi_1 \approx_{cl} \xi_2$

$\xi_1(x_1)$	$\boxdot$	0	$\boxdot$	5	2	0	…
$\xi_1(x_2)$	1	0	1	8	$\boxdot$	2	…
$\xi_1(y_1)$	$\boxdot$	0	$\boxdot$	5	2	0	…
$\xi_1(y_2)$	1	0	1	8	$\boxdot$	2	…

$\xi_2(x_1)$	0	5	2	0	…	
$\xi_2(x_2)$	1	0	1	8	2	…
$\xi_2(y_1)$	0	5	2	0	…	
$\xi_2(y_2)$	1	0	1	8	2	…

We see that $\xi_1 \approx_{fl}^{in} \xi_2$ but not $\xi_1 \approx_{cl} \xi_2 \Rightarrow$ it is not endochronous.

It has been proved by the example that Endochrony is not compositional.

Finally, the endochronous property allows the components work in an asynchronous environment.

The behaviour of the synchronous system is the same as the resynchronization.

Then, in the next chapter we are going to introduce in detail a set of distributes algorithms and to analyse their behaviour in synchronous system and after the process of desynchronization. If it is the case that both models share the same behaviour the components of the distributed algorithm are endochronous.

3 DESYNCHRONIZATION OF DISTRIBUTED ALGORITHMS

This chapter explains in detail each problem that matters in the field of distributed algorithms. Those small numbers of problem help us to cover the principal problem that we have to deal after the desynchronization and they have many different applications. Firstly, we consider working in the synchronous system and then, we desynchronize them and check if they are endochronous or not.

3.1 Detailed discussion for each algorithm

3.1.1 JOSEPHINE'S PROBLEM

We assume to have n women, each woman has one husband. They live in a village with their king. The king is honest and every people can trust on his information.
One day the king tells to n couples that there is at least one unfaithful man in the village. The king gives the order to kill the unfaithful man with a shot. The unfaithful man is killed by her wife at the end of the day. Every woman knows the fidelity of every husband except her own husband; however she is not allowed to designate the unfaithful husband in front of his wife. Moreover, she always listens to the shot.

The problem

The woman has to figure out if her husband is unfaithful or not.
In order to resolve the problem, we make some assumption:

- The king tells the truth.
- Every woman is: clever, thinks in the same way as the other women and obeys to her King. The woman i, $A_i \in \{A_1, A_2, \ldots A_n\}$ and the woman i´s husband, $B_i \in \{B_1, B_2, \ldots B_n\}$
- Any woman i shot her husband unless she certainly know her husband is unfaithful
- There are k unfaithful men. $1 \leq k \leq n$
- Every woman known the faithfulness of women´s husband except her own husband

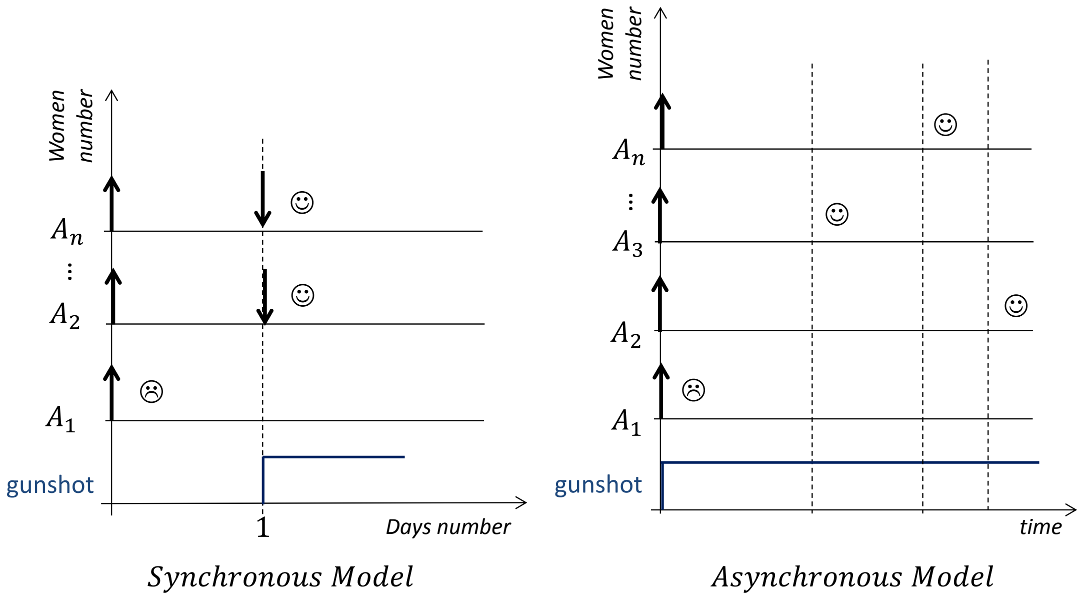
Solution for 1 unfaithful man

Let´s start with one unfaithful man, $k = 1$, we assume that B_1 is the unfaithful man. The woman 1, A_1, knows there is no unfaithful husbands. As a result, she instantaneously realizes her husband B_1 must be the unfaithful man. Consequently, she shoots her husband at the end of first day.

On the other hand, the woman i, $A_i \in \{A_1, A_2, \ldots A_n\} \wedge i \neq 1$, knows B_1 is unfaithful, but A_i is not sure if her own husband is faithful or not. Eventually, A_i listens to the shot and confirms her husband is faithful.

In order to receive the information "*listen to the shot at the day´s end*" we make two suggestions: the first one, where every women A_i receives the data at the same time, called synchronous model. And the second one, where the woman A_i does not necessary receives the data at the same time as the others women is the asynchronous model.

The case $k = 1$: A_1 has enough information to kill her husband. To be clearer, let see the following Figure 3.1.1.



↑	: A_i receives there is at least one unfaithful men
↓	: A_i expects to listen to $gunshot$
☹	: A_1 known her husband is unfaithful
☺	: A_i listen to $gunsho$t - her husband is faithful
gunshot	: A_1 kills her unfaithful husband, B_1

woman i, $A_i \in \{A_1, A_2, \ldots A_n\}$

Figure 3.1.1 Josephine´s problem for one unfaithful man

The algorithm works in both models. For the synchronous model, the $gunshot$is emitted on the first day, $day = 1$ same as number of unfaithful husband, $k = 1$. On the other hand, for asynchronous model, the$gunshot$ is emitted instantaneously, however the wife i will receives it at another point of time. It is show in the figure 1.

However, what happen if there are more than 1 unfaithful men, $k \geq 2$?

If there are more than one unfaithful man, the solution before explained does not work anymore. Then, we need to add other input as example the clock time, it is used in the Synchronous model.

Synchronous model

The process of receiving and sending the information is at the same time, in other words there is no a delay between any two processes, the communication is done instantaneously. Based on this model, every woman listens to the shot at the same clock time "clock time". The king said to kill the unfaithful husband at the end of the day, and then we consider each day as clock time.
Every day we synchronize all the inputs and the outputs of the woman i.

$\boldsymbol{Preposition}$: *At the end of k day, the woman $i \wedge i \in \{1,2,\dots n\}$ knows if her husband is faithful or not. Then the k unfaithful husbands are shot at the end of $day = k$.*
Then, the woman i needs the clock time to keep a tally of number of days.

$\boldsymbol{Proof}$: By induction

I. There are $k = 2$ unfaithful husbands are shot at the end of $k\ day$. Each end of the day is counted as a clock time, step.
   a. The unfaithful husband: $B_i\ \wedge i \in \{1,2\}$
      1. $A_i\ \wedge i \in \{1,2\}$ trust on her husband and she also knows there is at least one unfaithful husband, then $A_i \wedge i \in \{1,2\}$ expects to listen to at least one shot
      2. At the end of $day = 1$, any women listen the shots:
         Explanation:
         A_1 waits to listen to one shot emitted from A_2, and at the same time A_2 waits to listen to one shot emitted from A_1. Both women wait to listen to each other the first day but they do not.
      3. $A_i\ \wedge i \in \{1,2\}$ no listen to one shot at the end of $day = 1$, therefore, there must be one more unfaithful husband left. The unique option is: her husband is unfaithful. She shoots her husband the next day, $day = 2$. Then, $B_i\ \wedge i \in \{1,2\}$ are the unfaithful husbands
      4. $A_i\ \wedge i \in \{1,2\}$ makes the shot at the end of $day = 2$. After the woman A_i makes the shot, there are no unfaithful husbands left

   b. The faithful husband: $B_i\ \wedge i \in \{3,4,\dots n\}$
      1. $A_i\ \wedge i \in \{3,4,\dots n\}$ knows there are at least 2 unfaithful husbands, then She expects to listen to at least two shots
      2. $A_i\ \wedge i \in \{3,4,\dots n\}$ makes some assumptions about the $A_i\ \wedge i \in \{1,2\}$ ´s information has:
         - A_1 waits to listen one shot and A_2 waits to listen one shot
         - $A_i \wedge\ i \in \{1,2\}$ realizes her husband is unfaithful after not listening one shot at the end of the $day = 1$. She shoot her husband the next day, $day = 2$
         - At the end of $day = 2$, every woman i knows $B_i\ \wedge i \in \{1,2\}$ is unfaithful.

We prove this information before: $A_i \wedge\ i \in \{1{,}2\}$ shoots her husband at the end of $day = 2$

3. $A_i\ \wedge i \in \{3{,}4, \ldots n\}$ waits to listen two shots at the end of $day = 2$ and She does. Afterwards She confirms her husband is faithful

c. The two unfaithful husbands are shot at the end of the same day, $day = 2$, and the 2 shots are listened instantaneously by $\forall A_i \in \{1{,}2, \ldots n\}$

Then, we probe the number of unfaithful husbands is equal to the number of days:

$$k = day$$

II. We assume true for: $k = m \to$ At the end of m day, the woman i knows there are m unfaithful husbands. It implies the following:

a. Unfaithful husband: $B_i\ \wedge i = \{1{,}2, \ldots m\}$
$A_i\ \wedge i = \{1{,}2, \ldots m\}$ trust on her husband and knows there are at least $m - 1$ unfaithful husbands, then She waits to listen to $m - 1$ shots.
$A_i\ \wedge i = \{1{,}2, \ldots m\}$ waits to listen to $m - 1$ shots at the end of $day = m - 1$ but She does not. As a result, she realizes her husband is the unfaithful husband left. She make the shot at the end of next day, $day = m$.

b. Faithful husband: $B_i\ \wedge i = \{m + 1, m + 2, \ldots n\}$
$A_i\ \wedge i = \{m + 1, m + 2, \ldots n\}$ knows there are at least m unfaithful husbands, then She waits to listen m shots.
She expects to listen m shots at the end of $day = m$ and effectively she does. It allow her to confirm her husband is faithful

c. $A_i \wedge\ i = \{1{,}2, \ldots m\}$ makes the shot to $B_i \wedge\ i = \{1{,}2, \ldots m\}$ respectively at the end of $day = m$. Listen to the m shots, let the women i know there are m unfaithful husband and all of them are shot at the same time

d. $\forall A_i\ \wedge i = \{1{,}2, \ldots m\}, A_i$ listen to the m shots instantaneously at the end of the day, $day = m$

III. After assuming $k = m$ is correct. Let´s prove $k = m + 1$ is also correct.

a. Unfaithful husband: $B_i\ \wedge i = \{1{,}2, \ldots m + 1\}$

1. $A_i\ \wedge i = \{1{,}2, \ldots m + 1\}$ trusts on her husband and she also knows there are at least m unfaithful husbands. Then she expects to listen to m shots.
2. $A_i\ \wedge i = \{1{,}2, \ldots m + 1\}$ makes some assumptions from the information that $A_i\ \wedge i = \{1{,}2, \ldots m\}$ has:
It is assumed true for $A_i\ \wedge i = \{1{,}2, \ldots m\}$. It implies:
$A_i\ \wedge i = \{1{,}2, \ldots n\}$ listens to the m shots at the at the end of $day = m$
3. $A_i\ \wedge i = \{1{,}2, \ldots m + 1\}$ expects to listen m shots at the end of $day = m$ but she does not. Then, she realizes her husband is the one unfaithful left. She makes the shot the next day

4. The unfaithful husband i, $B_i \wedge i = \{1,2,\dots m+1\}$, is shot at the end of $day = m+1$ by his wife $i, A_i \wedge i = \{1,2,\dots m+1\}$ respectively. There is no unfaithful husband left.

b. Faithful husband: $B_i \wedge i = \{m+2, m+3, \dots n\}$
   1. $A_i \wedge i = \{m+2, m+3, \dots n\}$ trusts on her husband and she also knows there are at least $m+1$ unfaithful husbands. Then she expects to listen to $m+1$ shots from $A_i \wedge i = \{1,2,\dots m+1\}$
   2. $A_i \wedge i = \{m+2, m+3, \dots n\}$ makes some assumptions about the $A_i \wedge i = \{1,2,\dots m+1\}$´s information has:
      - $A_i \wedge i = \{1,2,\dots m+1\}$ is the wife of the unfaithful husband and We proved before that $A_i \wedge i = \{1,2,\dots m+1\}$ waits $m+1$ days to listen the $m+1$ shots
   3. $A_i \wedge i = \{m+2, m+3, \dots n\}$ waits to listen $m+1$ shots at the end of $day = m+1$ and She does. It allows her confirm her husband is faithful

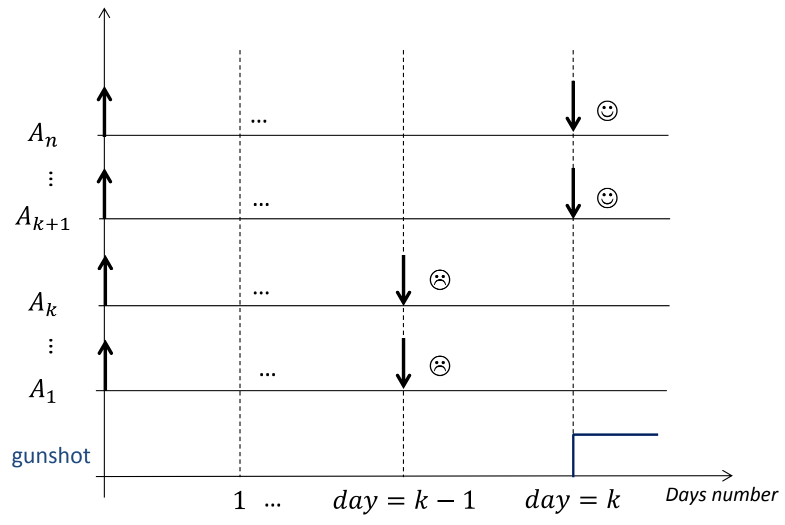
c. $A_i \wedge\ i = \{1,2,\dots m+1\}$ shoots her unfaithful husband $B_i \wedge\ i = \{1,2,\dots m+1\}$ at the end of $day = m+1$. In other words, The $m+1$ unfaithful husbands are shot at the same day

d. $\forall A_i \wedge\ i = \{1,2,\dots n\}$, A_i listen to the $m+1$ shots instantaneously at the end of $day = m+1$

We prove by induction that the number of days is equal to the number of unfaithful husbands:

$$day = k$$

The following figure explains how the synchrony algorithm works for Josephine´s problem.



↑	: A_i receives there is at least one unfaithful man
↓	: A_i expects to listen to $gunshot$
☹	: A_i does not listen to $gunshot$ - her husband is unfaithful
☺	: A_i listen to $gunshot$ - her husband is faithful
gunshot	: A_i kills her own unfaithful husband

Where:

woman i, $A_i \in \{A_1, A_2, \dots A_n\}$
unfaithful men number k, $1 \le k \le n$

Figure 3.1.2 Synchrony model for Josephine´s problem

We prove also that number of days is an important variable. It allows the women to keep a tally synchronously of the number of days and lets her realize the fidelity of her husband. The variable $day = clock\ time$ works as an input. It is predictable when the input comes and when the Josephine´s problem finishes.

Desynchronization of the problem – Asynchronous model

We have proved that the number of day is considered as an input variable that is needed for find out the unfaithful man. Therefore, we cannot remove it. As a result we can say that the Josephine´s problem does not work anymore after the desynchronization.

$\boldsymbol{Preposition}$: *Eventually,* $A_{\ i}$ *waits forever to listening to the shot.*

$\boldsymbol{Proof}$:

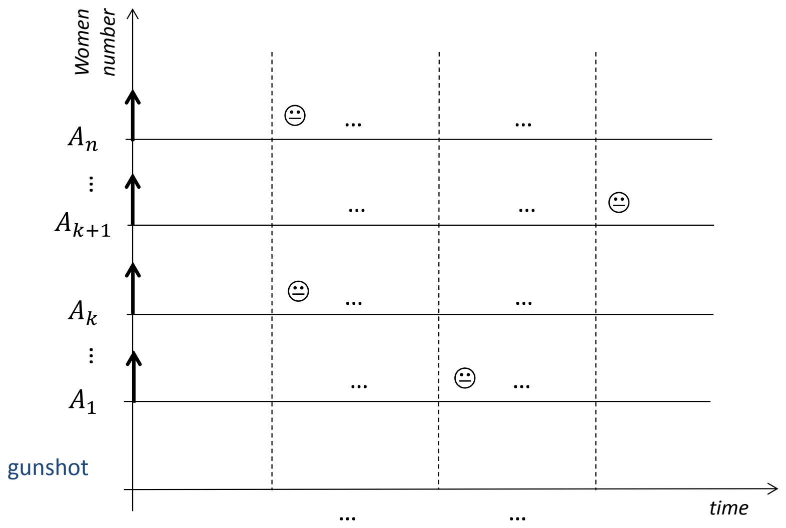
I. There are 2 unfaithful husbands. The assumption of the woman i, $A_i \wedge i \in \{1,2, \dots n\}$ is equal as in the synchronous model
   a. The unfaithful husband: $B_i\ \wedge i \in \{1,2\}$
      1. $A_i\ \wedge i \in \{1,2\}$ knows there is at least one unfaithful husband and $A_i\ \wedge i \in \{1,2\}$ expects to listen to at least one shot
      2. The shot is not emitted by the woman $A_i\ \wedge i \in \{1,2\}$
         Explanation:
         A_1 waits to listen to one shot emitted from A_2, but at the same time A_2 waits to listen to one shot emitted from A_1. There is no more information to let her know that her husband is the unfaithful husband left.
      3. Finally, $A_i \wedge i \in \{1,2\}$ never emit the shot to her unfaithful husband
      4. $A_i\ \wedge i \in \{1,2, \dots n\}$ could never know her husband is unfaithful.

   b. The faithful husband: $B_i\ \wedge i \in \{3,4, \dots n\}$
      1. $A_i\ \wedge i \in \{3, 4, \dots n\}$ knows there are at least 2 unfaithful husband. Then, $A_i\ \wedge i \in \{3, 4, \dots n\}$ expects to listen to two shots.
      2. $A_i\ \wedge i \in \{3,4, \dots n\}$ makes some assumptions about the $A_i\ \wedge i \in \{1,2\}$ ´s information has:
         - A_1 waits to listen one shot from A_2,and A_2 waits to listen one shot from A_1
         - $A_i\ \wedge i \in \{1,2\}$ does not shoot her husband unless she is certainly sure he is unfaithful, then she does not shoot
         - $B_i\ \wedge i \in \{1,2\}$ is unfaithful husband but there are not shoot
      3. $A_i\ \wedge i \in \{3,4, \dots n\}$ keep waiting

   c. $\forall A_i\ \wedge i \in \{1,2, \dots n\}$, A_i never listens to the shot and keeps waiting for ever. The $gunshot$ is never emitted.

The algorithm does not work for 2 unfaithful husbands. In general, it does not work for $more\ then\ two$ unfaithful husbands.

From the case I $\forall A_i \wedge i \in \{1,2,\dots n\}$ *, there is not possibility to know if the husband is faithful or not*. Let see the following picture.



↑	: A_i receives there is at least one unfaithful man
😐	: A_i does not listen to $gunshot$, She keeps waiting
gunshot	: A_i kills her own unfaithful husband

Where:

woman i, $A_i \in \{A_1, A_2, \dots A_n\}$
unfaithful men number k, $1 \leq k \leq n$

Figure 3.1.3. Asynchronous model for Josephine´s problem

After removing the notion of clock, the woman i does not have the perception of a deadline time that let her count. If the woman i cannot count, she cannot figure out if her husband is faithful. Therefore, the system does not work after the desynchronization.

However, there is another solution to make it work asynchronously. We implement an artificial input in each woman i, $whistle$.

The $whistle$ means:

The woman i blows a whistle to the woman $A_i \wedge i = \{1,2,\dots n\}$ in order to tell "$I\ am\ waiting, I\ do\ not\ what\ to\ do$".

The variable $whistle$ allows to the woman $A_i\ \wedge i = \{1,2,\dots n\}$ to have a counter, because each time she blows she make the end of her day. The variable $whistle$ is equivalent to the end of day that we have in the synchrony version.
At the end, at different points of time τ_i every woman knows if their husband is faithful or not.

Endochronous

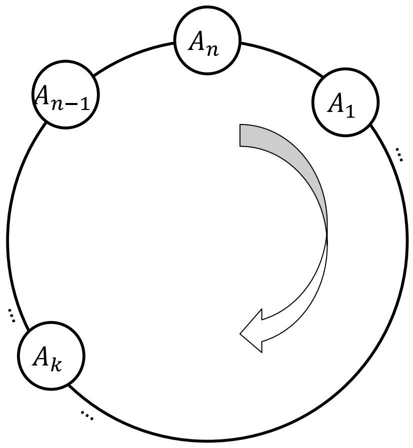
Without the notion of step, Josephine´s problem does not work. Therefore, it has not the endochrony property.

3.1.2 LEADER ELECTION IN A RING

We assume a ring network of n nodes. Every node A_i has the same characteristics, except its identifier token UID: t_i. Each node knows its clockwise neighbour, the node i can communicate through their token to the next node $(i+1)$. As a result, the token can move through the network in an unidirectional way.

Another specific characteristic is the identifier token must be an integer number and different from any other token. The identifier token is allowed to be manipulated by comparison.

The formal semantic we use is presented in the following figure:



Where:
Node i, $A_i \in \{A_1, A_2, \ldots A_n\}$ with their respective token t_i.
Token i, $t_i \in \{t_1, t_2, \ldots t_n\}$
Leader node $i = k$, A_k. If $1 \leq k \leq n$.
Non-leader node $i = j$, A_j . If $A_j = A_i \wedge j \neq k$

Figure 3.1.4. Ring network of n components

The Problem

We have to figure out who is the leader node.

The assumptions are:

- The ring is unidirectional and the token is manipulated by comparison as an integer number
- Each token is unique, there are not two igual tokens
- All the nodes know the number of nodes: n nodes
- The leader node k is the node that has the biggest token k: A_k and its token t_k

- Eventually, the non-leader j node must know they are not the leader: $A_k = \sup(\{\forall A_i\})$

The general solution

In each node i we make the comparison of the income token to its own token till get the leader:

- If the income UID token is greater than its own UID value, it keeps passing the token to the next node.
- If the income UID token is lower than its own UID value, it discards and does nothing.
- If the income UID token is equal than its own UID value, it means it is the $leader$.

During the next sentences, we use two words that they are necessary to highlight:

Cycle: A cycle means a point of time where the node receives the token and sends the token to his neighbour (Synchronous version).

Round: A round is when the token i has passed through all the nodes of the network, from $i = 1$ to $i = n$.

Firstly we are going to solve synchronously, then we are going to see if it works correctly in an asynchronous model. Finally, we explain which of both model suits better to the physical implementation. Let do the first model.

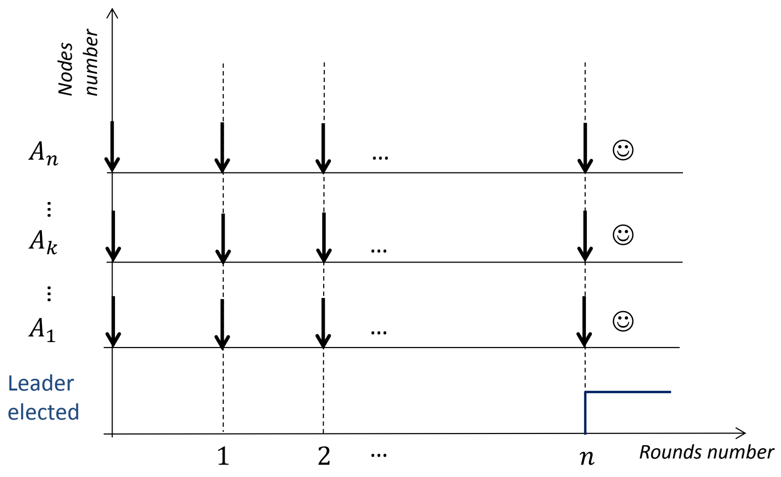
Synchronous model

Applying the general solution to the synchronous model, we have the following description:
All the nodes are indexed to the same clock time, being more precisely, the communication of every token is updated synchronously, in each cycle.

The process of receiving and sending is doing at the same time, which means when node i receive the token from the predecessor node $(i-1)$, it computes the comparison and sends instantaneously the result to the node $(i+1)$, A_{i+1}. In other words, the process of sending and receiving does not take time, it is immediate.

The behaviour of the synchronous model in the leader election is plotted in Figure 3.1.5.

As we explain before, the comparison between nodes give back three possible results. One of the results is “does nothing”, it informs the absence of a message and we treat as 0 value during the rest of leader election problem.



↓ : A_i receives the token from its predecessor neighbour A_{i-1}, compares and sends the result to its successor A_{i+1}

☺ : A_j knows A_k is the leader.

Where:

node network i, $A_i \in \{A_1, A_2, \dots A_n\}$
non-leader node j, $A_j = A_i \wedge j \neq k$
leader node A_k, $1 \leq k \leq n$

Figure 3.1.5. Synchronous model of Leader elected

Proposition: *After n rounds $\forall A_i$, node i knows node k is the leader.*

Proof: Case distinction:

I. The node k is assumed to be the leader $=>$ t_k is the biggest token in the ring network. Consequently, $\nexists\ t_j \geq t_k$. A_k knows himself it is the leader after n rounds.
   a. There are n nodes. Therefore the token t_k takes n cycles to reach to himself.
   b. In each comparison $t_k > t_j$, then it keeps passing to the next node till the number round n, when it compares to himself. Then, with n rounds the token t_k comes back to the node A_k.

   We deduce from both, the token t_k is the greatest identifier token, as a result the node A_k is the leader.

II. The node j is assumed to be the non-leader node, A_j. First we assume t_k takes r cycle to reach A_j. The value of r depend on k and j as we see in the following ecuation:

$$\text{cycle } r := \begin{Bmatrix} k-j & : if\ j > k \\ n-k-j & : if\ k < r \end{Bmatrix}$$

   a. At the cycle r, A_j receives t_k and pass it to A_{j+1}.
   b. After the cycle r, round $(r+1)$, A_j only receives the token $= 0$ .The token 0 is treated as nothing.

c. A_{j} keeps receiving token 0 after cycle $(r+1)$ till the cycle n.
d. If there is $t_{\mathrm{m}} > t_k$, t_m reachs A_{j} in r rounds. However it is not possible $\nexists\, t_{\mathrm{m}} > t_k$.

We deduce A_{j} knows A_k is the leader after the round n.

Pulling together the first and the second case, $\forall A_i$, A_i knows A_k is the leader after n rounds.

During the last proof, we assume that each node know how many nodes there are in the problem (n nodes), then each node knows if it is the leader or not after n rounds. However, if it is unknown by $\forall A_i$. How can the non-leader node realize that it is a non-leader node? How can we fit this requirement?

Implement a special input to every node when n is unknown

Implement a counter token in every node, the counter i of the node A_i counts the number of times the same token pass through its own A_i. In each comparison, the node is able to save in its memory the highest token, t_k. Consequently, at the end of the first round, it has in its memory the highest token, t_k.
We choose the special input because it needs less implementation: First, the leader node A_k realizes after receiving its own token. Second, it has to inform to every node A_j in order to let them know, otherwise $A_j$would never realize it. More precisely, A_k send to A_j a special token $LeaderElected$ in the second round.
Finally, the node A_i knows the node A_k is the leader after $round = 2 \times n$, due to the ring network.

The first n round let the leader node A_k knows it is the leader and the second round, $2 \times n$, let the node A_j knows it is not the leader. The leader node A_k sends a special token $LeaderElected$ to every node A_j, it will take n more cycles clock.

We have programmed in Quatz the LeaderElection in a ring of 3 nodes: The First program is for one node and the second program is the main program.

```
macro NoNodes = 3;
module Node(nat ?incomeUID, ?myUID,event LeaderReady,nat !send) {
     while(!LeaderReady) {// wait for leader elected
      if(incomeUID==myUID)
            emit next(LeaderReady);//is the LeaderELective
         else if(incomeUID<myUID)
            nothing;
         else
            next(send)=incomeUID;
         pause;    } }
```

```
macro NoNodes = 3; // number of nodes
module RingLeaderElected(event LeaderReady) {
    [NoNodes]nat arrayUID;
    [NoNodes]nat channelSend;
    arrayUID[0] = 5;
    arrayUID[1] = 2;
    arrayUID[2] = 1;
    for(i=0..NoNodes-1)
      channelSend[i] = arrayUID[i];

    for(i=0..NoNodes-1) do || let(i2 = (i==0 ? NoNodes-1 :i-1 ))
      Node(channelSend[i2],arrayUID[i],LeaderReady,channelSend[i]);
        pause;
    emit(LeaderReady);
}
drivenby {  await(LeaderReady); }
```

Desynchronization of the problem – Asynchronous model

Instead of having a global clock, every node has its own clock time. We remove Each node i works independently, the communication and computation are done asynchronously.
In synchronous model we use the time to keep synchronously updating the process. In each clock time, the node A_i receives and sends the token. However in asynchronous model, the node A_i sends the token to the next node A_{i+1}, but the node A_{i+1} receives it at time $\tau_j \wedge \tau_j > \tau_i$.

Proposition: *Eventually, the node* i *knows the node* k *is the leader.*

Proof: Case distinction:

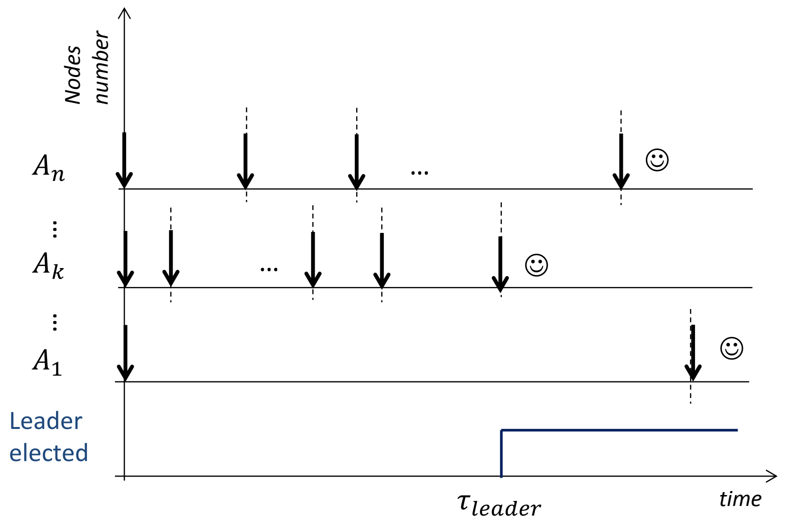
I. The node k is assumed to be the leader $=>$ t_k is the biggest token in the ring. Consequently $\nexists\, t_j > t_k$. A_k Knows himself it is the leader after the time τ_{leader} .
   a. In each comparison $t_k > t_j$ then, it keeps passing to the next node till the time τ_{leader}.
   b. After the time $\tau_{leader} =>$ t_k comes back to A_k.

   We deduce t_k is the greatest identifier token, as a result A_k is the leader.

II. Eventually, the non-leader node j knows A_k is the leader. A_j knows it is not the leader by receiving the special token $LeaderElected$
   a. If t_k takes r cycles to reach A_i. After the cycle $(r+1)$, the node A_i keeps receiving the token 0. This information is not significant.
   b. Since there is no deadline, clock time, A_i does not known when the leader is elected. Therefore, A_k must inform to every node A_j the leader is already elected.

c. The node A_i receives the token $LeaderElected$ from the node A_k, $time \geq \tau_{leader}$.

We deduce A_k must inform to every node A_j the leader is already elected.
Let see the Figure3, which show the behaviour of Leader Election in asynchronous model.



↓	: A_i receives the token from his predecessor A_{i-1}
☺	: A_i knows A_k is the leader

Where:

node network A_i, $A_i \in \{A_1, A_2, \dots A_n\}$
non-leader node A_j, $A_j = A_i \wedge j \neq k$
leader node A_k, $1 \leq k \leq n$

Figure 3.1.6. Asynchronous model of Leader Elected

From the case distinctions,$\forall A_i$, A_i eventually knows A_k is the leader .
A_i does not need to have information about the number of nodes of the network, it is an advantage from the synchronous model. However, A_j need to receive the special token $LeaderElected$.

Endochronous

We have proved that the leader elected can be described in synchronous model and asynchronous model as well. Both of them suits, therefore Leader Elected has the endochronous property. Let us see in detail the computation:

In each node:
- It is assume $a > b > c$, where $a, b, c \in \mathbb{N}$
- The stream of ξ_1 is for Non-Leader node and ξ_2 is for leader node
- The inputs: x_1 and x_2 are the income token and my token respectively
- The outputs: y_1 and y_2 are the higher token value and the leader token of the system

$Synchronous\ version$:

x_1	x_2	y_1	y_2
$(a :: A)$	$(b :: B)$	$[a]$	$[0]$
$(c :: A)$	$(b :: B)$	$[0]$	$[0]$
$(b :: A)$	$(b :: B)$	$[0]$	$[1]$

$\xi_1(x_1)$	2	7	0	8	…
$\xi_1(x_2)$	5	5	5	5	…
$\xi_1(y_1)$	0	7	0	8	…
$\xi_1(y_2)$	0	0	0	0	…
$\xi_2(x_1)$	2	7	0	8	…
$\xi_2(x_2)$	8	8	8	8	…
$\xi_2(y_1)$	8	8	8	0	…
$\xi_2(y_2)$	0	0	0	1	…

$Synchronous\ model\ (introducing\ \boxdot\ in\ the\ irrelevant\ values)$:

x_1	x_2	y_1	y_2
$(a :: A)$	$(b :: B)$	$[a]$	$[\boxdot]$
$(\boxdot :: A)$	$(b :: B)$	$[\boxdot]$	$[\boxdot]$
$(b :: A)$	$(b :: B)$	$[\boxdot]$	$[1]$

$\xi_1(x_1)$	2	7	$\boxdot$	8	…
$\xi_1(x_2)$	5	5	5	5	…
$\xi_1(y_1)$	$\boxdot$	7	$\boxdot$	8	…
$\xi_1(y_2)$	$\boxdot$	$\boxdot$	$\boxdot$	$\boxdot$	…
$\xi_2(x_1)$	2	7	$\boxdot$	8	…
$\xi_2(x_2)$	8	8	8	8	…
$\xi_2(y_1)$	8	8	8	$\boxdot$	…
$\xi_2(y_2)$	$\boxdot$	$\boxdot$	$\boxdot$	1	…

$Desynchronization$:

x_1	x_2	y_1	y_2
$(a :: A)$	$(b :: B)$	$[a]$	E
A	$(b :: B)$	D	E
$(b :: A)$	$(b :: B)$	D	$[1]$

$\xi_1(x_1)$	2	7	8	…	
$\xi_1(x_2)$	5	5	5	5	…
$\xi_1(y_1)$	7	8	…		
$\xi_1(y_2)$	…				
$\xi_2(x_1)$	2	7	8	…	
$\xi_2(x_2)$	8	8	8	8	…
$\xi_2(y_1)$	8	8	8	…	
$\xi_2(y_2)$	1	…			

We can see that after the desynchronization the behaviour of the stream does not differ from the synchronous system. Therefore, it is endochronous.

The synchronous model in the leader election is based on the comparison process which is instantaneous, it takes time $\delta = 0\ seconds$. The process of receiving and sending the token is done every clock time by every node, A_i, there is no notion of delay between the process of receiving and sending, it is ideal. However, it could not be implemented physically in real world.

Now we can take the advantage of endochrony and work in an asynchronous environment. It makes the system to run as fast as each node clock allow it, we could say that we improve the efficiency of the system. The leader peer A_k transfers the token k among the non-leader nodes A_j without taking care the non-leader time clock . It means t_k does not depend on any specific time clock.

3.1.3 LEADER ELECTION IN A GENERAL NETWORK

We have proved in the last problem that leader election has the same behaviour before and after the desynchronization, therefore here we are going to see if it works in a general network too. Now, the network is arbitrary and strongly connected and we figure out the shortest path to know the leader of the system.

Problem

The processes have a unique UID (identifier token) and they communicate to its neighbour in each step by sending first its own identifier and after the second round, make the same computation as leader lection in a ring network. Eventually, one process should be the leader by changing a special status component to the value Leader.

After that, there are several versions in the detail:

- It might be required that all non-leader process eventually output the fact they are not the $Leader$, by changing their status components to $no - Leader$
- The number of node,n, and the diameter, $diam$ can be either known or unknown to the processes.

Our algorithm requires that all the processes know $diam$, but it is not necessary the number of nodes of the network, n. The $diam$ show us the maximum number of steps until get to know the $Leader$.

The computation is:

- If the income UID token is greater than its own UID value, it keeps passing the token to the next node.
- If the incomes UID token ist lower than its own UID value, it discards and does nothing.
- If the incomes UID token is equal than its own UID value, it means it is the $leader$.
- In each step, each node stores in its local variable the maximum token, $maximumUID_i$
- After $diam$ rounds, all the processes store the $maximumUID$, then each of them know who the leader is. Due to that, they output the state of $leader$ or $non - leader$

Synchronous model

We consider the same semantic as in the leader election in a ring network.

$\boldsymbol{Proposition}$: *After $diam$ rounds* $\forall A_i$*, node i knows node k is the leader*.

$\boldsymbol{Proof}$:
Case distinction

I. The node k is assumed to be the leader $=>$ t_k is the biggest token in the network. Consequently, $\nexists\, t_j \geq t_k$.
   a. In each comparison $t_k > t_j$, then it keeps passing to the next node till the number round $diam$.
   b. At the round $diam$, the node i check the value stored in the variable $maximunUID$. If it is the same as its token, it is the $Leader$.

   We deduce, the token t_k is the greatest identifier token, as a result the node A_k is the leader.

II. The node j is assumed to be the non-leader node, A_j. First we assume t_k takes r cycle to reach A_j. The value of r depend on k and j as we see in the following ecuation:

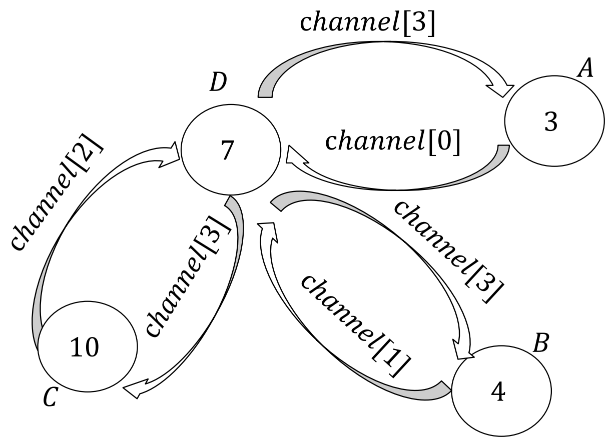
$$\text{cycle } r := \begin{Bmatrix} k-j & : if\ j > k \\ n-k-j & : if\ k < r \end{Bmatrix}$$

   a. At the cycle r, A_j receives t_k and pass it to its outgoing node or nodes.
   b. After the cycle r, round $(r+1)$, A_j has already stored in the maximum token in its local variable $maximumUID$. Therefore, during the next cycles until the cycle round it keeps sending the token 0.
   c. If there is $t_\mathrm{m} > t_k$, t_m reachs A_j in r rounds. However it is not possible $\nexists\, t_\mathrm{m} > t_k$
   d. At the round $diam$, the node i check the value stored in the variable $maximunUID$. If it is not the same as its token, it is the $non - Leader$

   We deduce A_j knows A_k is the leader after the round $diam$

From both cases*, we probe that all the processes know their state ($leader$ or $non - leader$) after round diam.*

An example: $n = 4$ nodes, the UID are $\{3,4,10,7\}$ and the $diam = 2$.



The program in QUARTZ is: generalNet is the main program, it has 4 nodes. The node D has three inputs and three outputs. The rest have one input and one output channel:

```
macro NoNodes = 4;
macro diam = 2;
module generalNet(event LeaderReady,event Lead) {
   [NoNodes]nat arrayUID;
   [NoNodes]nat channelSend;
   arrayUID[0] = 5;  arrayUID[1] = 2;  arrayUID[2] = 1;  arrayUID[3] = 10;

   for(i=0..NoNodes-1) { //initialization for channel sent
     channelSend[i] = arrayUID[i];  //max value sent to the neighbour
       }
     NodeA(channelSend[3],arrayUID[0],channelSend[0],LeaderReady);
    ||NodeA(channelSend[3],arrayUID[1],channelSend[1],LeaderReady);
    ||NodeA(channelSend[3],arrayUID[2],channelSend[2],LeaderReady);
    ||NodeD(channelSend[0],channelSend[1],channelSend[2],arrayUID[3],channelSend[3],LeaderReady,
Lead);
   pause;
   emit(LeaderReady);
}
drivenby {  await(LeaderReady); }
```

Node A, B and C:

```
//send: send the max value
//maxUID:Store the maxUID in each round
//LeaderReady: show the elective Leader is ready
macro NoNodes = 4;
macro diam = 2;
macro max(x1,x2) = (x1<x2 ? x2 : x1);
macro maxIncome(m,k) = (k==0 ? max(m,income[0]) : maxIncome( max(m,income[k]),k-1) );
module NodeA(nat ?incomeUID, ? myUID,nat !send, event Leader) {
   int i;
   [2]nat income;  int maxUID;  income[0] = incomeUID;  income[1] = myUID;
   pause;
   while(i<=diam) {
              next(i)=i+1;
              maxUID = maxIncome(0,1);
              next(send) = maxUID; //send maxUID
              if(i==diam & maxUID==myUID)//output leader
              {  emit next(Leader);           }
              pause; //step
   }       }
```

Node D:

```
macro NoNodes = 4;
macro diam = 2;
macro max(x1,x2) = (x1<x2 ? x2 : x1); // compute maximum of m and income[0..k]
macro maxIncome(m,k) = (k==0 ? max(m,income[0]) : maxIncome( max(m,income[k]),k-1) );
module NodeD(nat ?incomeUID1, ?incomeUID2,?incomeUID3,?myUID,!send,event Leader, event! Lead) {
   int i;
   [4]nat income;
   int maxUID;

   income[0] = incomeUID1;  income[1] = incomeUID2; income[2] = incomeUID3; income[3] = myUID;
   // in each round, the algorithm sends the maximum value: maxUID,
   // and in each round i must be incremented
      while(i<=diam) {
              next(i) = i+1;
              maxUID = maxIncome(0,3); // compute maximum of income[0..2]
              next(send) = maxUID; //send max
              if(i==diam & maxUID==myUID)//output leader
              {  emit next(Leader);   emit next(Lead);}
              wwhile:pause;
   }         }
```

Desynchronization -Asynchronous model

The difference between being in a ring network and being in the general network is that: In the general network we have a local variable where it saves the maximum value after the computation. And the other difference is that each node can have more than one inputs/outputs variable.

Each node compute:

- The inputs: $x_1, \dots, x_a$ the income tokens and x_2 my own token.
- It is assumed that $a > b > c$, where $a, b, c \in \mathbb{N}$. Consider the variables: $a = max\{x_1, \dots, x_a, z_1\}$, $c = max\ \{x_1, \dots, x_a, z_1\}$
- The local input variable z_1
- The outputs: y_1 and y_2 are the higher token value and the leader token of the system respectively
- The stream of ξ_1 is for Non-Leader node and ξ_2 is for leader node

After having explained the computation, it will be analogous to the leader election in a ring network.
Therefore the leader election in a general network is endochronous.

3.1.4 COORDINATE ATTACK

We explain here the importance of having a reliable communication between any two peers, without it the communication in the network does not work anymore. Therefore, we study a basic consensus problem based on the presence of communication failures called the coordinate attack problem.
The Coordinated attack problem is a fundamental problem of reaching consensus in a setting where messages may be lost.

The problem

There are several generals, who are located in different places. They want to attack a common known objective. They know that the only way to succeed is if all generals attack at the same time. Therefore, they have to reach a consensus of attack or not.

The generals can communicate with each other only by messengers who travel on foot. The messenger carries the information about the time attack or no time attack. However, the messenger may be captured or lost during the route and then the message may be lost. As there is no other way to communicate between the generals, they have to handle in order to get an agreement on whether to attack or not. The last statement is to attack if it is possible.

We suppose the following:
- There are n generals, $G_i \in \{G_1, G_2, \ \dots G_n\}$, and each general has one messengers who carries the message
- The communication is undirected and connected

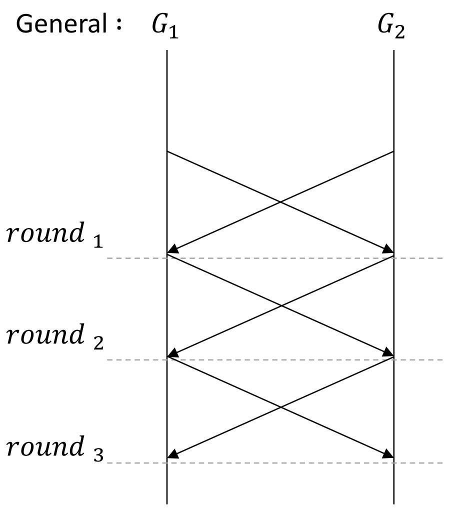
Proposition: *If there is at least one failure during the communication, the communication is broken whether it is synchronous or asynchronous.*

It is assumed that each general plans to attack, therefore each general send to the other generals the message: I am planning to attack at point t. Moreover, each general is considered as node or processor G_i, and the message as links.

Synchronous model

As each general is controlled by the same global clock, they wait to receive something at the end the clock time, even if it is not reliable.

We analyse the communication between 2 generals, G_1 and G_2, see the following figure:



$Round$ 1: G_1 sends the message to G_2 and vice versa, G_2 sends the message to G_1

$Round$ 2: G_2 receives the information from G_1 and sends the confirmation of having received it, so does G_1

$Round$ 3: G_1 receives the confirmation from G_2, and respond of having received the confirmation, so does G_2

$G_i \in \{G_1, G_2\}$ keeps sending messages to the other generals till the time of attack come up.

Figure 3.1.7. Reliable synchronous communication between two generals

We realize that if there is no communication failure, every general would attack the objective at the point t and being successful.

On the other hand, what happens if there is at least one message lost or manipulated? See the following figure. The first round is done correctly, but in the second round there is a communication failure, there are two possibilities:

- The messenger has been lost, G_2 does not received the message. G_2realize that the messenger must be captured and the communication is broken
- The messenger has been captured and the message would be manipulated, G_2 receives a wrong message. In the third round, G_2 would receive a different message, it let the general realize that the messenger must be captured, the communication is broken

Whether being any of them, G_2 goes into a conflict. The communication is broken between both peers. See the following figure:

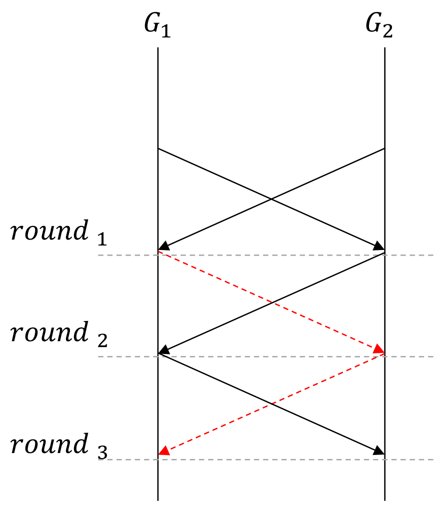


Figure 3.1.8. Unreliable synchronous communication between two generals

We showed that even one communication failure damages all the system.

Desynchronization

On the other hand, if each general have a different clock, our model behaves asynchronously. The argumentation is the same as the synchrony.

If there is no communication failure, all the generals eventually have the same information and attack the objective at the agreed time. However, if there is at least one failure during the communication, there are two possibilities as in the synchronous model, the communication is broken. See the following figure:

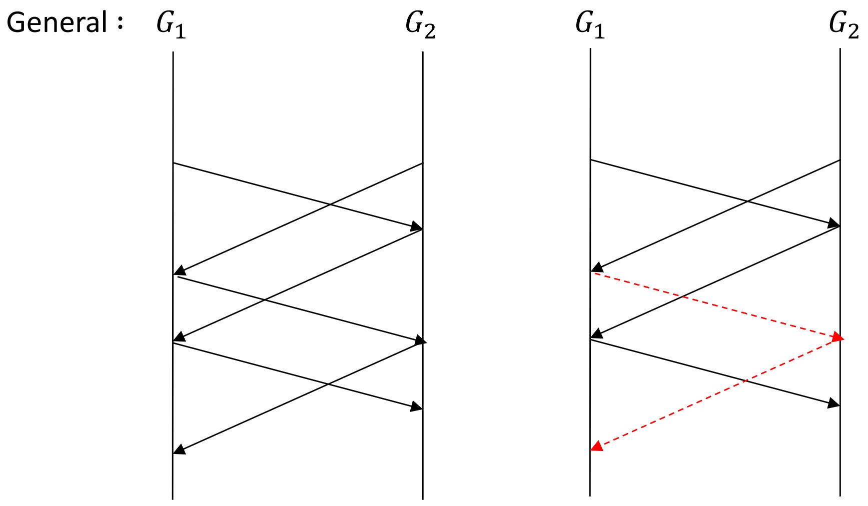


Figure 3.1.9. Asynchronous communication between 2 generals a) reliable links b) Unreliable links

We demonstrate that even the simpler communication between peer to peer could not be done correctly if there is at least one communication failure. Henceforth, the communication is supposed to be reliable.

3.1.5 STOPPING FAILURE

We have seen the failure during the communication of the message, but what happens if the failure is founded in the process. Now, we analyse what happens if the process does not work properly. Firstly, there are two failure models: the stopping failure problem, where the process may stop without warning and the byzantine problem, where faulty process may exhibit completely unconstrained behaviour.

We focus on the Stopping failure:

The problem

In the stopping failure model, at any point of time during the execution of the process, it may simply stop. Even, it can stop in the middle of a message sending step. Then, the other processor would receive part of the original message or nothing.
As in the coordinate attack problem, the nodes or processes want to reach a consensus agreement. We assume that the links are perfectly reliable; all the messages that are sent are delivered.

We have the following features:

- The network have n nodes, connected undirected graph with the other nodes
- Each process, $G_i \in \{G_1, G_2, \ \dots G_n\}$, knows the entire graph
- $G_i \in \{G_1, G_2, \ \dots G_n\}$ starts with an input from a fixed value set $V_i \in \{V_1, V_2, \ \dots V_n\}$. Each set V is composed by at least one element

Synchronous model

$\boldsymbol{Proposition}$: *If at least one node fails, it would propagate its message to the other nodes.*
Eventually each process verifies if it has received the correct message by checking the special variable, $decision$.

$\boldsymbol{Proof}$:
Firstly, we explain the algorithm for the synchrony model and it make us to understand better and prove informally the state before.

$The\ algorithm$:

In the initial state each node starts with an agreed value $v_j = \{1\}$ and a default value $v_0 = \{8\}$. If node G_i stop working, it sends a different value $v_j \neq \{1\}$.

In each round, the process G_i sends its own message to the other processor $G_j \wedge i \neq j$. At the same time the processor G_i receives the message and it stores it in a local variable after making the comparison. If the income token value is the same as it has, it does not store it; otherwise if it stores the element in the received set V_i.

After $(f + 1)$ rounds, the process checks whether the messages is correct or not: If the received set V_i has more than 1 element, it outputs $decision = v_0$, otherwise it outputs $decision = v_1$. The $decision = v_1$ shows that any process fails during the communication.

If one node fails during a particular round, it would propagate its message to the other nodes. After $f + 1$ rounds, the other nodes would receive it and add to the set value V. It makes us realize that even one node fail, it will break the reliable communication.

We program the problem of Stopping failure with two nodes, A and B. It has two failures in 3 rounds. The main program is Stopping failure and Node A and B are subprograms:

```
macro NoNodes = 2;
macro failure = 2; //maximum number of failure
macro v0 = 8; //default value for all nodes

module stoppingfailure(event LeaderReady) {

  [NoNodes]nat arrayUID;   [NoNodes]nat channel;   [NoNodes]nat Decision;
  arrayUID[0] = 1;   arrayUID[1] = 3;
  for(i=0..NoNodes-1) {
    channel[i] = arrayUID[i];
    Decision[i] = 0;
    }
  sfnodeA(channel[1],arrayUID[0],channel[0],Decision[0],LeaderReady);
   ||sfnodeB(channel[0],arrayUID[1],channel[1],Decision[1],LeaderReady);

  pause;
  emit(LeaderReady);
}
drivenby {   await(LeaderReady);}
```

Node A is called SFNODEA and Node B is called SFNODEB.

```
macro NoNodes = 2;
macro failure = 2; //maximum number of failure
macro v0 = 8; //default value for all nodes

module sfnodeA(nat ?income,nat ?myUID,nat
!outcome,nat !Decision, event !rdy) {
    int i;
   [NoNodes]nat storage1;
   next(i) = i+1;
   next(outcome) = myUID; //send
   storage1[0] = myUID;
   next(storage1[1]) = (income==myUID ? 0 :
income);
   pause;

   next(i) = i+1;    pause;

   next(i) = i+1; // do not send    pause;

  next(i) = i+1;
   if(i==failure+1) //end of the algorithm
   {
   next(Decision) = (storage1[1]==0 ? myUID : v0);
   emit next(rdy);
   }
   pause; //step
```

```
macro NoNodes = 2;
macro failure = 2; //maximum number of failure
macro v0 = 8; //default value for all nodes
//macro v = 1; //value of the node A
//macro singleton = 1;

module sfnodeB(nat ?income,nat ?myUID,nat
!outcome,nat !Decision, event !rdy) {

   int i;
   [NoNodes]nat storage;

   next(i) = i+1;
   next(outcome) = myUID; //send
   storage[0] = myUID;
   next(storage[1]) = (income==myUID ? 0 : income);
   pause;

   next(i) = i+1;
   next(outcome) = myUID;
   next(storage[1]) = (income==myUID ? 0 : income);
   pause;

   next(i) = i+1;
   next(storage[1]) = (income==myUID ? 0 : income);
   pause;

  next(i) = i+1;
   if(i==failure+1) //end of the algorithm
   {
   next(Decision) = (storage[1]==0 ? myUID : v0);
   emit next(rdy);
   }
   pause; //step
 }
```

As we see in Figure 3.1.10.There is no need that each process G_i work synchronously, it received and spread its message to the other processes. Eventually, after $(f+1)$ rounds, number of sending/receiving token, the process analyses its set V_i and output $decision$ as in the synchrony model.

We prove whether it is asynchrony or synchrony model, if the system has f failure process in total, each process would realize after the $(f+1)$ round.

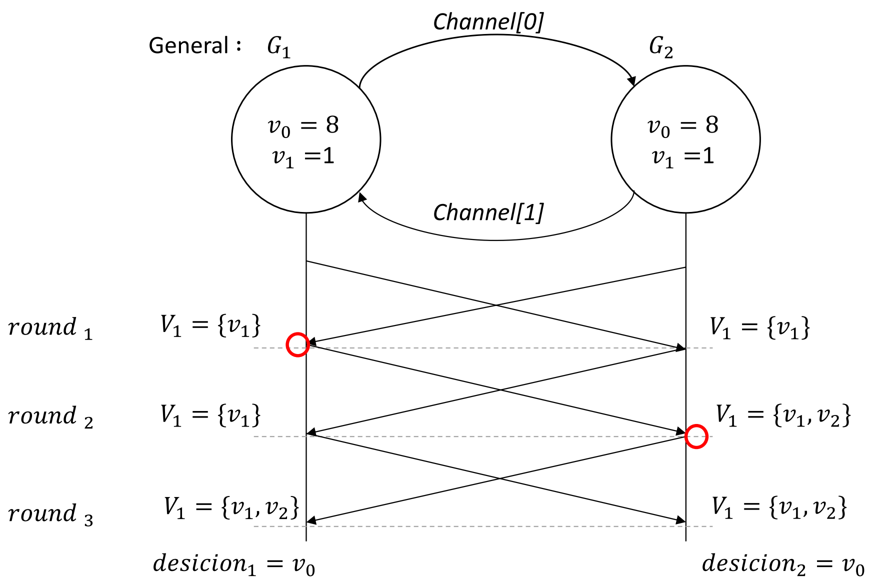


Figure 3.1.10. The communication between peer to peer has $f = 2$ failure processes. At the round 3, $f + 1$, all the process are affected.

The incorrect behavioural process misleads the other process about the information of the message. Consequently, the communication of the entire system falls. For this reason, it is important to have a correct behavioural process.

3.1.6 DINNING PHILOSOPHER

Informally, this problem deal with resources and users, the users need the resources at a certain point of time and depend on the availability it could get it for a while. Now, let us go to the theory and details of the problem.

The system has many resources that are shared among users. Here we explain how to use such specification to define resource –allocation problems. There are two different ways to solve a resource allocation: the explicit resource specification and exclusion specifications.

Example of explicit resource specification:

Consider 4 users: $U_1, \ldots U_4$ and the resources: $\{r(1), r(2), r(3), r(4)\}$

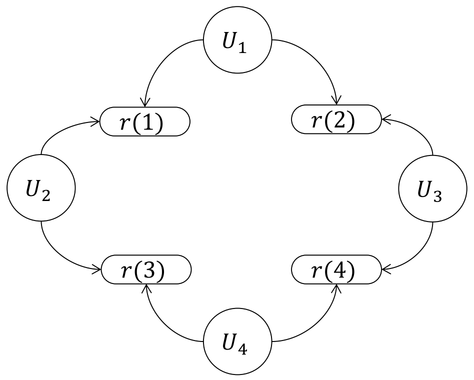


Figure 3.1.11. Example of explicit resource specification

It means, the U_1 needs exclusive the resource $r(1)$ and $r(2)$ to perform its work and for the others is the same. We can see here that U_1 and U_2 need the same resource $r(1)$, it is a conflict that we have to analyse, the same situation happens with the other users.

Example of exclusion specification:

It does not mention the resource, it count the users that are not allowed to use the resource. There are 4 users and consider the exclusion specification of the two elements sets $\{1,2\}, \{1,3\}, \{2,4\}$, and $\{3,4\}$. We note that U_1 does not exclude U_4, it means that they can perform their work simultaneously and the same with U_2 and U_3.

From both resource specifications, we choose the explicit resource specification because it is more general.

Before describe the problem we highlight the following characteristic that it is assumed during the solution of the problem: Resource allocation problem

We explain how to use the explicit resource specification into the resource-allocation, it is solved by share memory systems. We use the combination of user automata and a shared memory system automaton.

Besides that, we assume the trace properties: the well-formedness, exclusion, progress, independent progress and lock-out freedom condition. Each of this trace properties P has a signature consisting of $try, crit, exit$ and rem. See the following book for more information in chapter 10, 11 of LynchBook [2].

The problem

Related to the architecture we use a combination of user automata and share memory system automata. The cycle of one process is thinking and critical ($crit\ or\ eating$) regions.
It will be formulated in terms of explicit resource specification. There are n philosopher U_i seated around the table, between each pair of philosophers is a single fork (resource). There are n resources as well. The philosopher need to use two forks for having lunch, eating ($crit$). Therefore, from time to time, the user asks for the availability of both resources (right and left fork). When the philosopher becomes hungry, it seeks and it may attempt to eat. After eating the philosopher relinquishes the two forks.

Synchronous system

Firstly, we can informally discuss that there is no symmetric solution for the Dinning philosopher problem. If one user U_i and its neighbour seek to have lunch at the same time, at least one of them should eat ($crit$) while the other must wait until the other release the shared fork. Besides that, if all the users want to eat at the same time, each of them will take his left fork and the right fork will be already taken by its neighbour. The system in deadlocked, and there is no way to progress. Therefore, it is necessary to break the symmetry of the Dinning philosopher.

In order to resolve the conflict between two users, we implement an arbitrator. It will decide who goes to the critical state or waits. We explain it by one example:

The users U_1, U_2, U_3 are controlled by the arbitrator, each of them have their own state machine. The user U_i eats when the arbitrator gives the order, otherwise it will keep thinking.

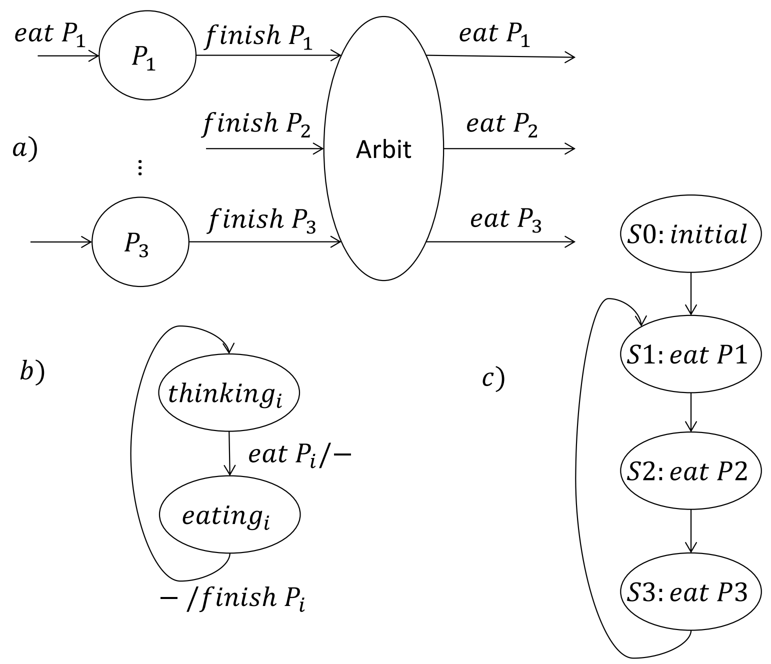


Figure 3.1.12. State machine of the $a)$ philosopher U_i and $b)$ the arbitrator

There is just one rule on the firing rule belongs to the philosopher, then just one way to compute, therefore it is endochronous.

Synchronous version:

Firing rule for Philosopher: *Firing rule for the Arbitrator*:

x_1	y_1
$(a :: A)$	$[a]$

x_1	x_2	x_3	S	S'	y_1	y_2	y_3
$(a :: A)$	$(b :: B)$	$(c :: C)$	$S0$	$S1$	$[1]$	$[0]$	$[0]$
$(a :: A)$	$(b :: B)$	$(c :: C)$	$S0$	$S1$	$[1]$	$[0]$	$[0]$
$(1 :: A)$	$(b :: B)$	$(c :: C)$	$S1$	$S2$	$[0]$	$[1]$	$[0]$
$(a :: A)$	$(1 :: B)$	$(c :: C)$	$S2$	$S3$	$[0]$	$[0]$	$[1]$
$(a :: A)$	$(b :: B)$	$(1 :: C)$	$S3$	$S1$	$[1]$	$[0]$	$[0]$

After the Desynchronization:

Firing rule for Philosopher: *Firing rule for the Arbitrator*:

x_1	y_1
$(a :: A)$	$[a]$

x_1	x_2	x_3	S	S'	y_1	y_2	y_3
A	B	C	$S0$	$S1$	$[1]$	$[0]$	$[0]$
$(a :: A)$	$(b :: B)$	$(c :: C)$	$S0$	$S1$	$[1]$	$[0]$	$[0]$
$(1 :: A)$	$(b :: B)$	$(c :: C)$	$S1$	$S2$	$[0]$	$[1]$	$[0]$
$(a :: A)$	$(1 :: B)$	$(c :: C)$	$S2$	$S3$	$[0]$	$[0]$	$[1]$
$(a :: A)$	$(b :: B)$	$(1 :: C)$	$S3$	$S1$	$[1]$	$[0]$	$[0]$

Moreover, the firing rule belong to the arbitrator have not any conflict between them too, therefore it is endochronous.

With this example we informally prove that the Dining philosopher work properly after the desynchronization.

3.1.7 BREADTH-FIRST SEARCH

The problem of Breadth-first search is motivated by the need to build structures suitable for supporting efficient communication. Here, we perform how to establish a breadth-first spanning tree for the digraph. The motivation for constructing such a tree comes from the desire to have a convenient structure to use as a basis for broadcast communication.

The problem

We define a direct spanning tree of a direct graph $G = (V, E)$ to be rooted tree that consist entirely of directed edges in E. All the edges directed from parents to children in the tree, and that contains every vertex of G.
We assume that:

- The network is strongly connected and there is always a source node P_0
- All the process except i_0 should have a parent process
- The communication is done over directed edges

Synchronous model

First, we are going to explain what happens in each process i:
There is some set of process that is marked, initially P_0. Process P_0 sends a *search* message at round 1, to all of its neighbours. At any round, if an unmarked process receives a *search* message, it marks itself and chooses one of the processes from which the *search* message has arrived as its parent. At the first round after a process gets marked, it sends a *search* message to all of its outgoing neighbours.

We can see that this algorithm produce a tree. We can prove that after r rounds, every process at distance d from P_0 in the graph has its parents pointer defined. We assume also a reliable communication and non-suspicious behaviour in each node.

Let see with an example how works the algorithm. There are 4 nodes that are strongly connected as shown in the figure:

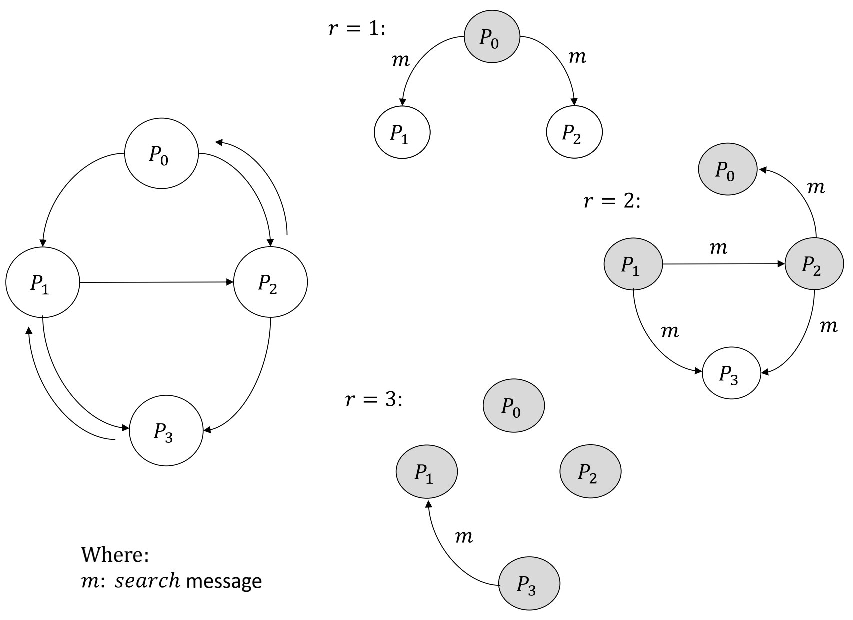


Figure 3.1.13. Synchonous process to solve breadth-first search

We want to broadcasts a *search* message to all the nodes. It needs 3 rounds to communicate the message. Each node has its parent, except the source node P_0. As we are working in the synchronous model:

- P_1 has only receive one message and it is from P_0. Then it is its parent. The same case for P_2, even it will receive later another message from P_1.
- P_3 has to choose from P_1, P_2 as its parent.

We have to stress the behaviour of the node P_3, in synchrony system it receives from its preprocessors P_1, P_2 the message at the same time. Afterward, it chooses between both nodes as its parents. There are 2 kinds of spanning trees are:

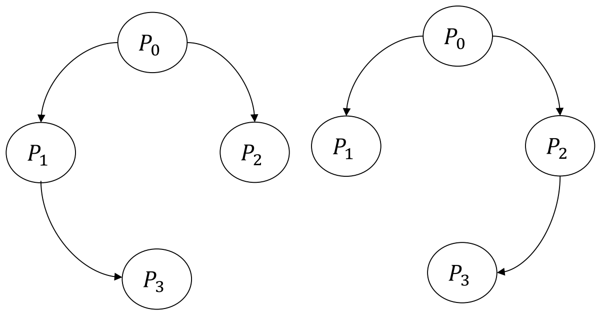


Figure 3.1.14. Synchronous spanning tree

Desynchronization

The problem comes when the node has to choose between various predecessor nodes as its parent. The node chooses between them in synchronous system, however not in an asynchronous system. It means, that depend on the arrived time message to decide which of them its parent is.

Therefore, we can see that if we desynchronize the system it will become an asynchronous system. In the asynchronous system, we do not have the notion a global clock any more.
In the asynchronous system, the node P_3 does not have to choose between the predecessors because one of them will arrive faster than the other. After making the resynchronization of the asynchronous system, the spanning tree may be totally different from the synchronous system. In fact, there is a new more design of the spanning tree.

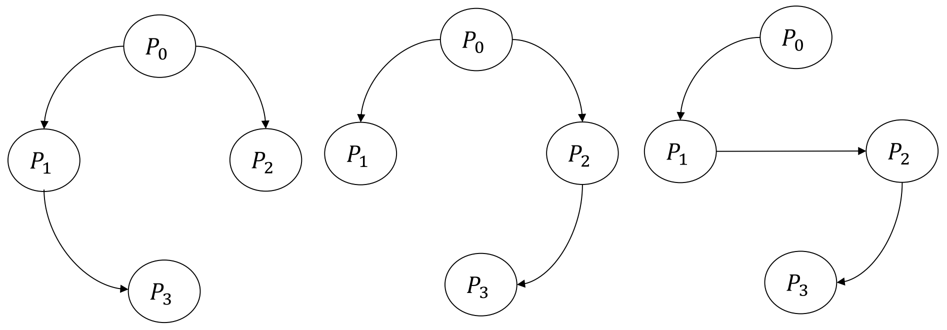


Figure 3.1.15. Aynchronous spanning tree

Due to the breadth first search is not endochronous, we have to implement a new design to make it work in an asynchronous network. See the next chapter Synchronizer.

3.2 Summary of each algorithm

After having all the problems in detail, let us make a table where we can point the endochrony property out.

Problem	Endochronous	Synchronizer
Josephine´s problem	✗	✓
Leader election in a ring network	✓	Does not need
Leader election in a general network	✓	Does not need
Stopping failure	✓	Does not need
Dinning philosopher	✓	Does not need
Breath-first search	✗	✓

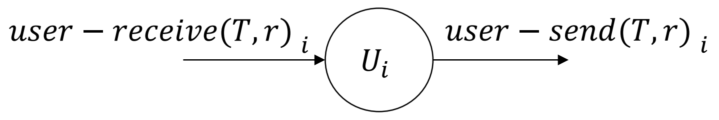
4 ENDOCHRONY AND SYNCHRONIZER

If the problem does not fulfil the endochronous property, we have to find a way to make the components work in an asynchronous environment. Therefore, the Synchronizer is a good idea to achieve this work. The synchronizer is a system module that transforms the synchrony model to the asynchrony model. There are many types of synchronizers that are detailed explained in [2] and all of these implementation involve synchronizing the system at every synchronous round, the implementation allow to work for arbitrary synchronous algorithms.

We begin with the Global synchronizer that specified the correctness in term of I/O automata. Then we define the local synchronizer abstractly and show that it implements the global specification. Moreover, we assumed that there is no failure during the communication or during the computation.

$The\ notation$:

In synchronous system each process i is presented as a kind of state machine, with message generation and transition functions. Here, we modified by representing each process i as a "user process" I/O automaton U_i. We understand better with the example below.
Then, we define the tagged message to be pair (m, i), where $m \in M$ and $1 \leq i \leq n$. The user automaton U_i has output action of the form $user - send(T, r)_i$, where T is a set of tagged messages and $r \in \mathbb{N}^+$, represent the round number, those are sent to its neighbours. Moreover, U_i has input action of the form $user - receive(T, r)_i$ by which receives the message from its neighbours. U_i perform the $user - send(0, r)_i$ when it has not any message to send at round r.
Here, we module the inputs and the outputs of the user automata using input actions rather than encoding them in states, see the following figure.



Where:
T: indicate the message resource
r: round number

Figure 3.2.1. I/O User automaton

We explain the theory thought an example: $user - send$ and $user - receive$ actions.

We suppose that we have 4 nodes, $n = 4$. Then the $user - send(\{m_1, 1\}, \{m_2, 2\}, 3)_4$ indicates that at round 3, user U_4 sends message m_1 to user U_1 and m_2 to user U_2 and sends no other messages. Also, $user - receive(\{m_1, 1\}, \{m_2, 2\}, 3)_4$ indicate that at round 3, U_4 receives message m_1 from user U_1 and message m_2 from user U_2, and receives no other messages.

U_i is expected to preserve the *well-formedness condition* (see more detail in [2]) that the $user - send_i$ and $user - receive_i$ actions alternate, starting with a $user - send_i$ action, and that successive pairs of actions occurs in order of rounds. The sequence of such actions is a prefix of an infinite sequence of the form: $user - send(T_1, 1)_i, user - receive(T'_1, 1)_i, user - send(T_2, 2)_i, user - receive(T_2, 2)_i, \ldots$

The other action is the *liveness condition*. U_i must eventually perform a $user - send()_i$ at round r such that $user - receive()_i$ events for all previous rounds have already occurred. U_i keeps sending messages for infinite many rounds as long as the systems keeps responding.

Now, we are ready to describe the rest of the system as a $Global\ synchronizer$. The job of the global synchronizer is at each round collect all the messages that are sent by user automata at that round in $user - send$ actions and deliver them to all the user automata in $user - receive$ actions. It synchronizes globally, after all the $user - send$ events and before all the $user - receive$ events of each round.
The figure 4.1. shows the combination of user U_i and $Global\ synchronizer$ automata, those make the *Global Synchronizer system.*

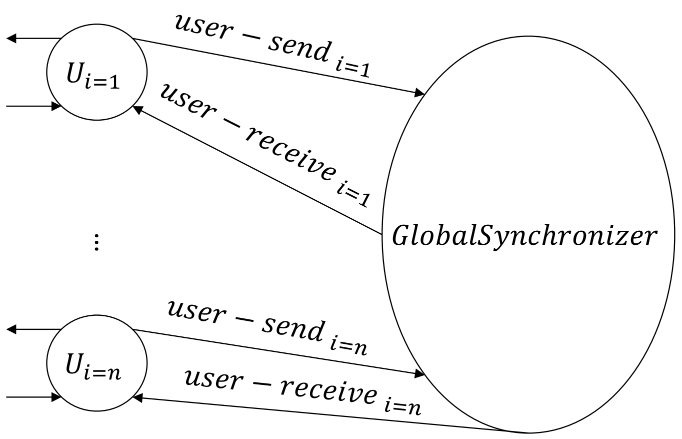


Figure 3.2.2. Architecture of the *Global Synchronizer system*

We can see that any algorithm in synchronous network model can be described in this new style: a composition of user automata U_i and the $Global\ synchronizer$ automaton.

Finally, the $synchronizer$ implement the $Global\ Synchronizer$ automaton with an asynchronous network algorithm, with one process P_i at each node i of the graph G and a reliable FIFO send/receive channel $C_{i,j}$ in each direction on each edge (i, j) of G.

We could say finally that the user U_i cannot note the difference between running in the synchronous system and running in the $Global\ Synchronizer$ system.

Local synchronizer

The $local\ synchronizer$ involves synchronization among neighbours rather than among arbitrary nodes. This advantage saves time and communication complexity. The only difference between $Global\ synchronizer$ and $local\ synchronizer$ is in the action $user - receive$:

- It is not necessary to wait for messages form all users in the entire network as in $synchronizer$. We just need to wait for the neighbour messages, as soon as in round r messages can be sent to U_i it can receive from all its neighbours.

We have seen before that Josephine´s problem and Breath-First Search do not fulfill the endochronous property. Therefore, we are going to implement in each of them a $synchronizer$.

Here, we explain how would work the problem of Breath-First Search, because it is more intuitive to understand than Josephine´s problem. The breadth-first search is a clear example for using a $local\ synchronizer$. Between each node and its outgoing processor there is a $local\ synchronizer$.
We use the same example that as before but introducing the new design:

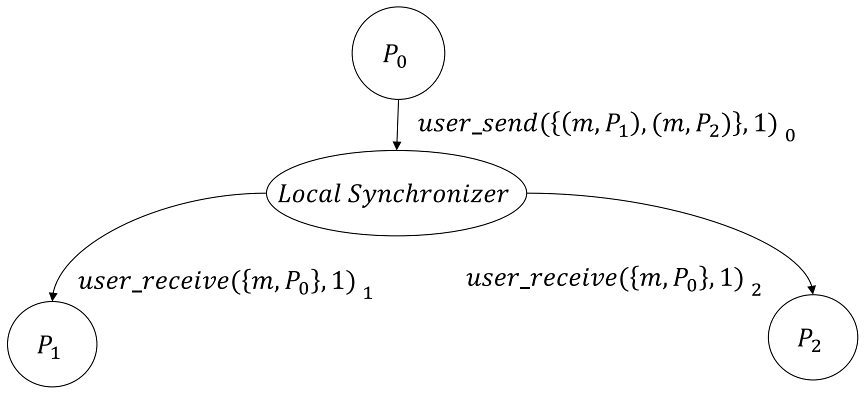

P_0
$user\_send(\{(m, P_1), (m, P_2)\}, 1)_0$
$Local\ Synchronizer$
$user\_receive(\{m, P_0\}, 1)_1$
$user\_receive(\{m, P_0\}, 1)_2$
P_1
P_2

Figure 3.2.3. Local synchronizer in the breadth-first search

We can intuitively see that $local\ synchronizer$ makes the same spanning tree as the synchronous system.

5 SUMMARY

In the past, we used to work with synchronous systems where all its components work together and trigger at the same time. Those synchronous systems were not really complex. However, we nowadays find really complex systems that make really difficult to control by just one global clock. Therefore, instead of having one control clock the system were divide in simpler systems. Each simpler system has its own local clock and communicates with the others simpler systems asynchronously. When we speak system we are taking about embedded system and hardware system design.

On the other hand, the synchronous system is well state due to the long time research. It has many tools to verify the correctness of the design. It simplifies programming, since developers do not have to take care about low-level detail like timing, synchronization and scheduling. The distributed systems are asynchronous by nature and the asynchronous systems are difficult to simulate and verified due to the asynchronous concurrency. Those are reason for not remove all the synchronous system. Instead of removing, we are going to reuse them in an asynchronous environment. Therefore, we need to adapt all the systems in the new interface. In order to reuse the synchronous system in the asynchronous network (distributed system) we make the desynchronization of the synchronous system. However, first we have to prove if the synchronous system is able to desynchronize in a correct form, endochronous property.

Then, the endochronous property permits the components of a synchrony system work in an asynchronous environment. In other words, those components take advantages of the synchronous models for simulation and verification. Besides that, they communicate independently to each other.

Finally, we have studied the endochrony of some distributed algorithms that have many applications in real system for example finding a leader process in the network, making a consensus agreement in the network, share the resource between the processes, broadcast a message to other process in a short path, ... Moreover, we take into account the consequences of having a failure during the communication or in the process.

After having studied distributed algorithms, some of them do not fit the endochronous property, for example the Josephine´s problem and the Breadth first reach. Those algorithms need to implement a synchronizer in order to make them work in an asynchronous system. In summary, all the distributed algorithms explained are prepared to work in an asynchronous environment.

BIBLIOGRAPHY

[1] Klaus Schneider, “From synchrony to asynchrony” - Hardware and software system, Department of Computer Science, University of Kaiserslautern, January 2014.

[2] Nancy A. Lynch, “Distributed algorithms”, Morgan Kaufmann Publishers. In San Francisco – California – USA, 1996.

[3] Klaus Schneider, “The synchronous programming language Quartz”, A Model-Based Approach to the Synthesis of Hardware-Software Systems, Department of Computer Science, University of Kaiserslautern, 2010.

[4] D. Potop-Butucaru, B. Caillaud, and A. Benveniste, “Concurrency in synchronous system”, In application of Concurrency to System Design (ACSD), pages 67-76, Hamilton, Ontario, Canada, 2004. IEEE Computer Society.

[5] J. Pantaleone, “Synchronization of Metronomies”, American Association of Phisycs Teachers, 70(10): 992-1000, October 2002.

[6] A. Benveniste, B. Caillaud, and P. Le Guernic, “From synchrony to asynchrony”, In J.C.M. Baeten and S. Mauw, editors. Concurrency Theory (CONCUR), volume 1664 of LNCS, pages 162-177, Eindhoven, The Netherlands, 1999. Springer.

[7] Edward A. Lee, and Thomas M. Parks, “Dataflow process network”, IEEE volume 85, number 5, 1995.

[8] M. K. McClintock. “Menstrual synchrony and suppression”, Nature, 229: 244-245, 1971.

[9] Z. Neda, E. Ravasz, Y. Brechet, T. Vicsek, and A. L. Barabasi, “Self-organization in the concert hall: the dynamics of rhythmic applause”, Technical report arXiv: cond-mat/0003001, arXiv.org, http:/arvix.org, 2000.

[10] J. Buck, “Synchronous rhythmic flashing of fireflies”, the Quarterly Review of Biology, 63(3):265-289, 1988.

[11] V. Torre, “A theory of Synchronization of two heart pacemaker cells”, Journal of Theoretical Biology, 61:55-71, 1976.

[12] E.A. Lee and Santiovanni- Vicentelli. “Comparing models of computation”. In International Conference on Computer-Aided Design (ICCAD), pages 234-241, San Jose, California, USA, 1996. ACM/IEEE Computer Society.

[13] Jens Brandt, Mike Gemünde and Klaus Schneider, “Desynchronizing synchronous programns by modes“, Application of Concurrency to System Design (ACSD), 2009, IEEE Computer Society.

[14] Brandt and K. Schneider and Y. Bai, “Passive Code in Synchronous Programs”, Transactions on Embedded Computing Systems (TECS), 2014

[15] A. Benveniste, L.P. Carloni, P. Caspi, and A.L. Sangiovanni-Vincentelli, “Heterogeneous reactive systems modeling and correct-by-construction deployment”, in R. Alur and I. Lee,

editor, Embedded Software (EMSOFT), volume 2855 of LNCS, pages 35-50, Philadelphia, Pensylvania, USA, 2003. Springer.