Fault Tolerance for Stream Programs on Parallel Platforms

by
Vicent Sanz Marco

A thesis submitted to the University of Hertfordshire
in partial fulfillment of the requirements of the degree of

Doctor of Philosophy

December 2015

Abstract

A distributed system is defined as a collection of autonomous computers connected by a network, and with the appropriate distributed software for the system to be seen by users as a single entity capable of providing computing facilities.

Distributed systems with centralised control have a distinguished control node, called leader node. The main role of a leader node is to distribute and manage shared resources in a resource-efficient manner. A distributed system with centralised control can use stream processing networks for communication. In a stream processing system, applications typically act as continuous queries, ingesting data continuously, analyzing and correlating the data, and generating a stream of results.

Fault tolerance is the ability of a system to process the information, even if it happens any failure or anomaly in the system. Fault tolerance has become an important requirement for distributed systems, due to the possibility of failure has currently risen to the increase in number of nodes and the runtime of applications in distributed system. Therefore, to resolve this problem, it is important to add fault tolerance mechanisms order to provide the internal capacity to preserve the execution of the tasks despite the occurrence of faults.

If the leader on a centralised control system fails, it is necessary to elect a new leader. While leader election has received a lot of attention in message-passing systems, very few solutions have been proposed for shared memory systems, as we propose.

In addition, rollback-recovery strategies are important fault tolerance mechanisms for distributed systems, since that it is based on storing information into a stable storage in failure-free state and when a failure affects a node, the system uses the information stored to recover the state of the node before the failure appears.

In this thesis, we are focused on creating two fault tolerance mechanisms for distributed systems with centralised control that uses stream processing for communication. These two mechanism created are leader election and log-based rollback-recovery, implemented using LPEL.

The leader election method proposed is based on an atomic Compare-And-Swap (CAS) instruction, which is directly available on many processors. Our leader election method works with idle nodes, meaning that only the non-busy nodes compete to become the new leader while the busy nodes can continue with their tasks and later update their leader reference. Furthermore, this leader election method has short completion time and low space complexity.

The log-based rollback-recovery method proposed for distributed systems with stream processing networks is a novel approach that is free from domino effect and does not generate orphan messages accomplishing the always-no-orphans consistency condition. Additionally, this approach has lower overhead impact into the system compared to other approaches, and it is a mechanism that provides scalability, because it is insensitive to the number of nodes in the system.

Abstract

Los sistemas distribuidos se definen como una conjunto de ordenadores conectados por red, y con el apropiado software son vistos por los usuarios como una simple entidad capaz de trabajar como un unico ordenador.

Sistemas distribuidos con control centralizado tienen un nodo control specifico, llamado nodo lider. El objetivo principal del nodo lider es la distribucion y gestion the recursos compartidos de una manera eficiente. Un Sistemas distribuidos con control centralizado puede usar stream processing para la communicacion. En sistemas stream processing, las applicacion tipicamente actuan como queries continuas, ejecutando data continuadamente, analizando y relacionandola, y generando stream con los resultados.

La tolerancia a fallos es la habilidad de un sistema para acceder a la informacion, incluso si ocurriese un fallo or anomalia en el sistema. Por esta razon, la tolerancia a fallos se ha convertido en un importante requisito para sistemas distribuidos, debido a que la posibilidad de fallo ha crecido actualmente por el incremento del numero de nodos y la applicacion ejecutadas usados por los sistemas distribuidos. Por lo tanto, para resolver este problem es imporante aadir mecanismo de tolerancia de fallos que tienen capacidad interna para preservar la ejecucion de las tareas a pesar de la aparicion de fallos.

Si el lider falla, el sistema necesita elegir a un nuevo lider para poder continuar. Mientras la eleccion de lider ha recibido mucha atencion en sistemas de paso de mensajes, muy pocas soluciones han sido propuestas para sistemas con memoria compartido.

Las estrategias de rollback-recovery son importantes mecanismos de tolerancia de fallos para sistemas distribuidos, ya que se basan en almacenar informacion en una memoria estable cuando no hay fallos en el sistema y cuando un fallo afecta un nodo, el sistema isa la informacion alamacenada para recuperar el estado del nodo antes de la aparicion del fallo.

En esta tesis, nos enfocamos en crear dos mechanismos de tolerancia de fallos para sistemas distribuidos con control centralizado que usan stream processing para comunicarlos. Los mecanismos creados son la eleccin de lider y log-based rollback-recovery, implementados usando LPEL.

El mtodo de eleccin de lider propuesto se basa en la instruccin Compare-And-Swap (CAS). Nuestro mtodo trabaja incrementando, significa que solo los nodo libres actualizan su referencia de lider. El mtodo log-based rollback-recovery propuesto para sistemas distribuidos con stream processing es una nueva aproximacion que esta libre del domino effect y no genera mensajes huerfanos cumpliendo la condicion the always-no-orphans. Adicionalmente, nuestro mecanismo tiene un bajo coste dentro del sistema y permite la scalabilidad, porque el nmero de nodos en el sistemas no es importante para nuestro mecanismo.

Acknowledgements

I would like to show my deepest gratitude and appreciation to my extra supervisory team who were willing to support me throughout my PhD studies. The encouragement and inspiration given to me by my principal supervisor Dr. Raimund Kirner is second to none. He was able to identify my weakness early on and gently encourage and support these areas of my personal development. He has always shown a deep interest in my work and was always willing to talk about it. My second supervisor Dr. Michael Zolda has also provided me with exceptional support. He has the ability to make me view ideas from a different perspective which helped the overall development of this work. I will always appreciate what they have done for me.

I would like to thanks my friends and colleagues in the research institute and school of computer science.

Finally, I would like to show my appreciation to my family and my fiance Olga, for their never-ending love and always encourage me to work hard and to value education.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"Most of the fundamental ideas of science are essentially simple, and may, as a rule, be expressed in a language comprehensible to everyone."

— Albert Einstein

High performance computing is currently supported by distributed systems with hundreds of processors [91]. The demand for higher-performing of scientific applications is met by further increasing the number of components. Nevertheless, the risk of faults increases. Although the average time between faults (Mean Time Between Faults, MTBF [125]) of single system components is high, the system may often fail because of the large number of components it has, such as, power supplies, fans or network boards [115].

A distributed system consists of a collection of autonomous computers linked by a computer network and equipped with distributed system software [124]. The distributed system software enables computers to coordinate their activities and to share the resources of the system: hardware, software, and data. Users of a distributed system should perceive a single integrated computing facility, even though it may be implemented by many computers in different locations. Benefits of distributed systems include bridging geographic distances, improving performance and availability, maintaining autonomy, cutting costs, and allowing for interaction.

There are different types of distributed systems [79]. However, this thesis focuses on distributed systems with shared memory [17]. A distributed system with shared memory is composed of nodes with a local memory that is available only for itself, and shared variables that can be read/written by all nodes in a single shared memory. This kind of architecture is used by High Performance Computing (HPC) machines such as Archer [50] or Tianhe-2 [103] or Intel Paragon multicomputer [37].

One of the main characteristics of a distributed system is the notion of partial faults [10]: part of the system fails while the remaining parts continue performing and seemingly correctly. An important goal in a distributed system design is to construct the system in such a way that it can automatically recover from partial faults without seriously affecting overall performance. Whenever a fault occurs, the system should continue to operate in an acceptable fashion while repairs are underway. In other words, a distributed system is expected to offer **fault tolerance**.

Fault tolerance is the system's ability to continue working and accessing information, even in the event of a fault or anomaly [15]. For this reason, fault tolerance is important for some kinds of distributed systems, such as storage systems or air traffic control systems, in which fault tolerance is required to avoid catastrophic consequences of faults. Storage systems with fault tolerance are vital in environments that use critical information, such as financial institutions, governments and corporations.

The objective of designing and creating fault tolerance for distributed systems is to ensure that the system can continue to work properly as a whole, even in the presence of faults. The system needs to be able to recover from faults once they appear, and to remove the system's error status.

The disadvantage of fault tolerance techniques are the performance costs as it is an added task during execution times. This execution time overhead is called overloading [89], and lots of works on this topic have been published [110, 121, 113].

In this thesis, a system consists of a set of hardware and software components, and it is designed to provide a specific service. A system malfunction occurs when the system does not perform these services as specified. Then a system fails when it does not realize its specification. Therefore, we assume that a system's erroneous state is such that it could lead to a failure in the system.

This thesis proposes two fault tolerance mechanisms for distributed systems to repair the faults that affects the system with local hardware faults with transient and persistent state. When the leader node is affected by a fault, the leader election method presented in this thesis is called. When a node, that it is not the leader of the system, is affected by a fault, the log-based rollback-recovery mechanism presented is called. These mechanisms were chosen because it is not necessary to add new hardware to the system and can be creating given a low execution time overhead.

The first mechanism is for electing a leader (or master) that was affected by a fault and it is not working anymore, from a collection of processes in the presence of shared memory. Reaching a consensus about which process should become the leader is an essential fault-handling measure in systems that depend on a central master process. Master processes are often used to manage shared resources, or to distribute work over several worker processes.

The second mechanism is for taking snapshots of the streaming network state and it offers the possibility of recovering a previous state should one of the processes in the network fail. The rollback-recovery mechanism presented herein implements backward error correction.

In this introductory chapter, we provide an introduction of our leader election and rollback-recovery mechanism, and we also outline our contributions and the structure of this thesis.

1.1 Leader election

The idea of a leader node for coordination is common in distributed systems. However such a leader provides a single point of failure [92]. Therefore, mechanisms are required to make the leader fault-tolerant. For example, such mechanisms could detect a failed leader and then initiate leader election to replace the old leader. Leader election is about solving the consensus problem of choosing a unique leader among a set of nodes.

Many distributed system applications are based on the existence of one distinguished leader node that coordinates the work of a set of nodes [39]. If a system detects that the leader has crashed, then the system has to find a node in the system to become the new leader. The election of a new leader requires nodes reaching an agreement about which one will become the new leader. Numerous methods have been proposed to address the leader election problem in different system contexts [96, 42, 8].

In this thesis we propose a new leader election algorithm, which has been developed for distributed systems with shared memory, whose advantage is to recover the system from a faulty leader without external assistance. The proposed algorithm is based on the following preconditions:

1. The system provides a shared memory that is accessible by all nodes
2. The nodes of the system can implement the Compare-And-Swap (CAS) instruction

As a result, we obtain a distributed system with a fault-tolerant leader independently of the topology of the system and the communication protocol used by the nodes.

To create the prototype for the experiments, the proposed leader election is implemented in LPEL [99, 109]. In LPEL, there are two different roles: worker and leader. A leader is also called a master and there is only one leader in the system. Each worker represents a parallel execution thread. LPEL uses one worker for each core. Currently, LPEL does not offer a solution when the leader stops working. The leader stops, for example, if the core where the leader is performed crashes, or if communication between the leader and workers is cut by accident. So, the system collapses.

The aim of our prototype is to modify LPEL so that it can automatically recover after the leader fails or stops working. The leader is replaced with an available worker that becomes the new leader of the system. As a result, LPEL has a fault tolerance mechanism that affects only the leader of the system.

Section 3.1 summarizes the most leader election methods related to our context of distributed systems with shared memory.

1.2 Rollback-Recovery

The second proposed fault tolerance technique is a rollback-recovery mechanism for distributed systems.

A very basic rollback-recovery mechanism for fault tolerance consists in regularly storing a checkpoint of the system state while the system is free of faults. The moment when the system is storing this checkpoints is called *phase of protection* or *failure-free.* If a fault has been detected in the system, the restore mechanism reloads the state of the latest stored checkpoint and retries it again. The moment when the system is using the restore mechanism is called *phase of restoring.*

These protocols have been studied [78] and compared according to the time and resources used by the system in the phase of protection and recovery from the latest checkpoint.

The main overhead in rollback-recovery strategies in modern systems stems from storing the checkpoint [49]. Techniques like uncoordinated checkpointing help to reduce this overhead [53]. In the real-time domain, techniques like state prediction to make a *roll-forward* instead of a roll-back are used to avoid high timing overhead at the expense of approximative result [95].

The proposed technique is a log-based rollback-recovery that consists in restarting the node affected by a fault using the last checkpoint stored. After restarting the node, it uses the log information stored in its stable storage to restore its state to the state it had before the fault appeared.

The following list shows the assumption for the prototype created using LPEL according to our log-based rollback-recovery mechanism:

- The mechanism is only used by the workers and not the leader. Therefore, a fault to activate the restore mechanism can affect only workers.
- A fault cannot affect the messages between the leader and the worker(s). So, we are sure that the messages received from the leader and the worker are corrects without errors. We can assume this using, for example, the handshaking mechanism to ensure that the messages are correct [36].
- The workers cannot communicate with each other.
- Each worker and the leader have a stable storage. The stable storage of a worker/leader can be accessed only for itself.

This prototype is implemented only for the worker nodes of LPEL because the leader node has the proposed leader election mechanism. This prototype creates a new role for the leader called Fault-Tolerance Control (**FTC**).

As a result, we create a strong fault tolerance mechanism for LPEL along with both proposed algorithms (leader election and log-based rollback-recovery).

Section 3.2 summarizes the most rollback-recovery methods related to our context of distributed systems with shared memory.

1.3 Research Question

This thesis is motivated by the following research question:

Can log-based rollback-recovery be used as an adequate fault tolerance technique for stream processing networks with centralised scheduling on shared memory systems?

This question is fractured into the following sub-questions:

1. How can log-based rollback-recovery be optimised specifically for stream programs?

   overall processing model-role and operation of the scheduler

2. How can a distributed system with centralised scheduling be protected if the leader of the system is affected by a fault causing a single point of failure?

3. What fault model can be handled by the resulting log-based rollback-recovery approach?

1.4 Contributions

A fault tolerance system is one that has been designed to maintain an acceptable level of service in the presence of certain faults that have been anticipated by the developers. The precise definition of an acceptable level of service depends on the specific system application, and may range from an average service degradation threshold to the complete absence of service failures.

Fault tolerance in stream-processing systems is an important issue, as stream-processing applications are a frequent application pattern that is well-suited for programming parallel systems. However, there are not a lot of fault tolerance mechanisms developed for distributed systems that use stream processing networks.

In this thesis we contribute two mechanisms. The first mechanism is an efficient and fast fault-tolerant leader election method that converts a distributed system with shared memory into a fault-tolerant system. The second mechanism is a rollback-recovery mechanism (log-based rollback-recovery) to convert distributed systems with stream processing networks into a fault-tolerant system.

Regarding the leader election proposed, it is based on an atomic Compare-And-Swap (CAS) instruction, which is directly available on many processors. Our leader election method works with idle nodes, which means that only non-busy nodes compete to become the new leader, while busy nodes can continue with their tasks and later update their leader reference.

The other proposed method, log-based rollback-recovery, is used to convert streaming networks on parallel platforms into a system that can handle faults. The idea is to store the minimum information of specific moments of the application in log buffers. If the application fails, the system uses the log information that has been stored to rollback the task to the previous state before the fault appeared. According to this idea, the system is able to access information, even if a fault or an anomaly has taken place in the system.

In opposition to other rollback-recovery mechanism we consulted, that had a high execution time overhead, we created a new rollback-recovery mechanism that has a low execution time overhead in failure-free time. One of the reasons why we achieved this feature is through to the creation of a new garbage collection mechanism with low execution time overhead, not presented in the others mechanisms.

Another contribution of this thesis is to work toward adding fault tolerance methods to S-Net [61, 62, 100] and LPEL [109, 99]. With this idea, we intend to incorporate the proposed mechanism (leader election and log-based rollback-recovery) to LPEL. The addition of these methods to LPEL implies that S-Net becomes a fault-tolerant system that can handle faults. However S-Net has specific implicit tasks like *synchrocells* (explained in Section 2.2) that is not supported for our approach. As a result, the approach presented in this thesis supports only a static software-pipeline of stateless components.

The modification of LPEL is primed to benefit both programmers and compiler designers. Programmers stand to benefit from an automatic fault tolerance mechanism for which they do not require previous knowledge. Regarding compiler designers that uses LPEL to create a compiler, the compiler created will receive automatically the fault tolerance techniques added in LPEL. So, the compiler created will have both fault tolerance mechanism proposed.

Last but not least, as a member of the CRAFTERS [116] Project, the mission of our research group [102] was to add fault tolerance to LPEL. Therefore, adding these two fault tolerance techniques to LPEL has fulfilled the purpose of our group in the CRAFTERS Project.

1.4.1 Publications

The following paper within the scope of this thesis have been published:

- Vicent Sanz Marco, Michael Zolda, Raimund Kirner, *"Efficient Leader Election for Synchronous Shared-Memory Systems"*, In Proc. Int'l Workshop on Performance, Power and Predictability of Many-Core Embedded Systems, Dresden, Germany, March,2014

- Vicent Sanz Marco, Raimund Kirner, Michael Zolda, Frank Penczek, *"Fault-tolerant Coordination of S-Net Stream-processing Networks"*, In Proc. 2nd Workshop on Feedback-Directed Compiler Optimization for Multi-Core Architectures, Berlin, Germany, Jan. 2013.

Furthermore, there are two journal articles under preparation to be submitted:

- Vicent Sanz Marco, Raimund Kirner, Michael Zolda, *"A Fast and Fault-Tolerant Leader Election Algorithm for Shared Memory Systems"*

- Vicent Sanz Marco, Raimund Kirner, Michael Zolda, *"Log-Based Rollback-Recovery for streaming networks"*

1.5 Thesis Structure

The remainder of this thesis is organized as follows.

Chapter 2 provides some background for the fault tolerance context. This background provides a description of the features of faults and the classification of failures. It also offers an overview description about our leader election and log-based rollback-recovery methods by describing the overall approach and discussing potential alternative approaches. This chapter includes a summary of stream-processing networks and data-flow programming. It also provides a description of LPEL where prototypes are implemented.

Chapter 3 offers an overview of the related work by setting out the background of this research. This chapter primarily reviews research for the field of leader election models in distributed systems with shared memory. Additionally, this chapter also describes rollback-recovery strategies in distributed systems and streaming networks.

Chapter 4 presents the fault model that our fault tolerance mechanisms need to support. In addition, this chapter shows the reason why we chose the leader election and log-based rollback-recovery mechanisms.

Chapter 5 discusses the proposed leader election method. This chapter includes the description of the leader election preliminaries. Then it presents a full description and some examples of the proposed leader election algorithm. At the end of the chapter, an extensive example and the evaluation of the leader election method are presented in terms of space complexity and time complexity.

Chapter 6 introduces the proposed log-based rollback-recovery mechanism approach. This chapter includes the description of the proposed new role (FTC), the proposed garbage collection and the proposed restore mechanism for our log-based rollback-recovery method. There are some examples of the log-based rollback-recovery at the end of the chapter.

Chapter 7 demonstrates the correctness of the fault tolerance mechanisms presented in this thesis. This chapter includes the necessary definitions and the correctness that proves the fault tolerance and the correct execution of the approaches.

Chapter 8 presents the experiments of our leader election and log-based rollback-recovery method using LPEL. Moreover, it explains the notification message mechanism. This mechanism is implemented to know if the leader is still alive or if it is necessary to start the leader election mechanism. These experiments are tested for a distributed system with 4, 8, 16, 32 and 48 cores.

In conclusion, **Chapter 9** summarizes the work done, and discusses the strong points and the weak points of the proposed approaches. It also provides directions for future research in this field.

Chapter 2

Background

2.1 Fault tolerance

Fault tolerance is the property that enables a system to continue operating properly if a failure event appears in any system component [69]. Once the errors appear, the system needs a mechanism so it can recover and continue working independently of hardware or software errors. It is important to not confuse fault tolerance with system maintenance. The main difference is that system maintenance requires an external agent that has to work when a problem appears, whereas in fault tolerance an internal system property, allows its recovery.

A system's fault tolerance can be explained as a set of actions characterized by their ability to recognize any alteration to the process that the system is running on and is, therefore, potentially harmful [98]. This alteration is called a fault. This recognition is the necessary first step to organizing a response that aims to neutralize or eliminate that fault [89]. Moreover, we talk about fault tolerance once the error has occurred, and the system can stop or disable the error through an urgent action to overcome it with no external help. Therefore, fault tolerance recognizes the error, intercepts it urgently and handles it so that it can continue its usual work.

To some extent, fault tolerance reminds us of the immune system in the human body, which is characterized by its ability to recognize any foreign molecule in the organism (antigen) which is, therefore, potentially harmful. This recognition, called antigenic presentation, is the necessary first step towards organizing a response that aims to neutralize or eliminate that antigen. However, we should state that the response in the fault tolerance of a system has a faster and more effective response as it deals with the error immediately.

Fault tolerance related concepts in distributed systems have become increasingly important [83, 72]. The proliferation of distributed systems and their use in a growing number of areas means that finding a solution for any kind of error that may arise is urgent [85]. A large proportion of the available literature on fault tolerance refers to distributed systems. These works are divided into different areas according to where the faults occurs, such as processors, communications or data areas. In some applications, systems are very critical and need to be protected against failure. Incorrect system functionality can be catastrophic and can cause major problems in the system. For example, a failure that stops the flight-control system on a plane in the middle of a flight is a major problem that the fault tolerance mechanisms have to avoid. In addition, fault tolerance is important in systems that must work uninterruptedly for 24 hours a day without rest.

To be able to make precise statements about fault tolerance, it is important to clarify the meaning of the associated terms, like system, fault, error, and failure [12].

- A **system** is a set of interacting components that form a conceptual unit. A system's observable behaviour is called *output*. *Input* is external excitation through any well-defined interfaces.

- We call a **fault** any physical or logical defect in any component, hardware or software in a system. This category includes accidental contacts between electrical components, cuts in electrical components, defects in components, variations in component functionality due to external disturbances (such as temperature or electromagnetic waves), etc. We would say that a fault is due to physical phenomena.

- An **error** is the manifestation or the result of a fault. In other words, an error is the result of a fault from the system's information point of view. Errors form part of the called *informational world*.

- If an error causes malfunction of the system from an external point of view, this means that if the consequences of the fault go beyond the system, a **failure** occurs. Failures occur outside the system, which means that failures occur in the external world or user world.

To clarify these concepts and to better explain the difference between faults, errors and failures, we use a combinational circuit example. If one point of the circuit is connected incorrectly to the logic value 0, the system has a **fault**. If this fact is seen from the point of view of the circuit's truth table, we find an **error** since this table has changed as a result of the fault.

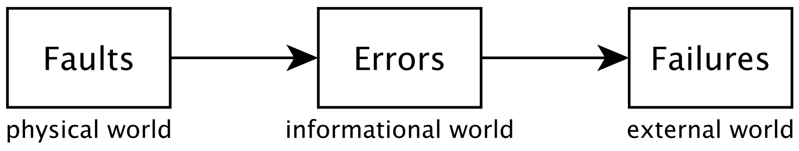


Figure 2.1. Faults, errors and failures

When this error affects the system's output or the performance of a hardware of the system, we have a **failure**. Figure 2.1 shows the relation among faults, errors and failures.

A *fault-tolerant system* is a system that has been designed to maintain an acceptable level of service in the presence of certain faults that have been anticipated by the developers. The precise definition of an acceptable service level depends on the specific system application, and may range from an average service degradation threshold to the complete absence of service failures. A *robust system* is a system that can maintain an acceptable service level in the presence of non-specific faults. Once again, the precise definition of an acceptable service level depends on the specific system application.

2.1.1 Features of faults

Before we consider mechanisms to support robustness and fault tolerance, it is important to think about what kinds of faults we wish to consider. There is no comprehensive approach that can handle all possible kinds of faults, so it is necessary to restrict ourselves to certain classes of faults. Different approaches may be necessary to handle different kinds of faults. Faults can be characterized by several criteria: cause, nature, duration, extension and variability [98]. Figure 2.2 depicts features of faults.

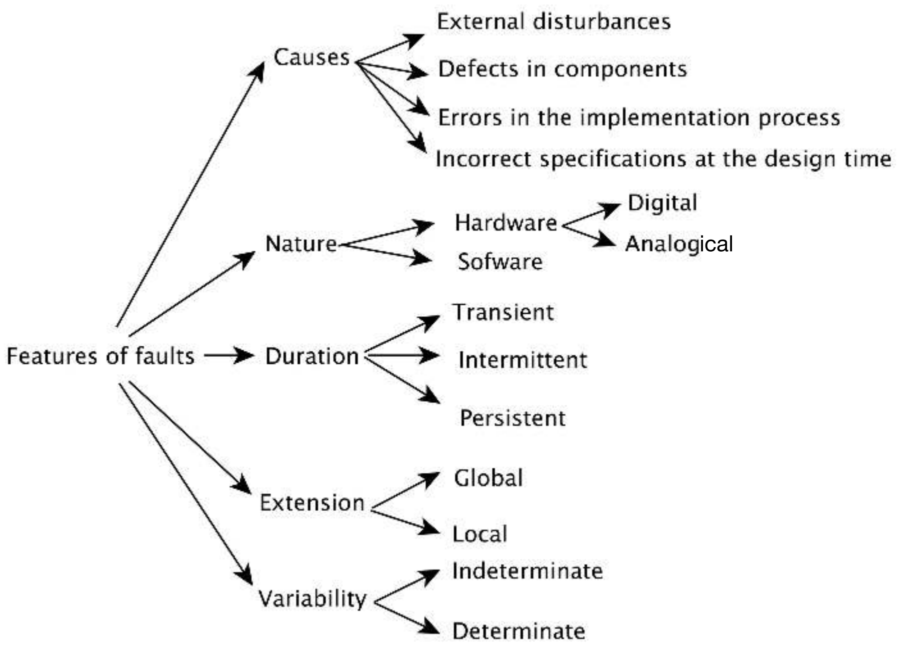


Figure 2.2. Features of faults [7]

Causes

The *causes* of faults can be many: incorrect specifications at the design time, errors in the implementation process, defects in components, external disturbances, etc.

Nature
: The *nature* of faults specifies the part of the system that fails: **software** or **hardware**. Inside hardware, the fault can be analog or digital.

Duration
: Regarding *duration*, faults can be **persistent**, **intermittent** and **transient**. Persistent faults are characterized by continuing indefinitely in time if there is no action to repair them. Intermittent faults appear, disappear, and reappear repeatedly and randomly. Transient faults appear only during brief moments and coincide with some circumstance, such as an external disturbance.

Extension
: The *extension* of the fault indicates if the fault affects only a localized point (**local**) or if it affects the totality (**global**) of hardware, software, or even both.

Variability
: In terms of *variability*, faults can be either **determinate**, if their state does not change with time, even if the input or other conditions change, or **indeterminate**, whose state can change when some conditions alter.

2.1.2 Classification of Failures

For economic and feasibility reasons, it is not useful to try and avoid all possible failures. Instead the system should only fail in such ways that the remaining service level is still acceptable for application. We consider the following list the for categorizing failures [64, 40]:

- Fail-silent failures
- Omission failures
- Timing failures
- Byzantine failures

Fail-silent failures
: A *fail-silent failure*, also called *fail-stop failure*, occurs when a node halts completely in the middle of an execution for no reason. This means that the node cannot send or receive messages. The system can detect the affected node using a specific detection mechanism to do so.

Omission failures

An *omission failure* is associated with communication protocols of distributed systems. This failure appears when a message is lost between the communication of two nodes and neither one of these two nodes knows that the message has been lost. So the receiving node fails to receive incoming messages or the sending node fails to send outgoing messages. As a result, the sending node does not know if the sent message was received correctly or the receiving node does not know if there has been a message.

Timing failures

Timing failures are related with the violation of a temporal property in synchronous distributed systems or real-time systems. For example, there are two nodes, n_1 and n_2, and their clocks are not synchronized. If n_1 sends a message to n_2, n_2 receives the message but, in the point of view of n_2, the message has been delayed longer than a threshold period. Thus n_2 does not accept the received message and a timing failure appears in the system.

Byzantine failures

The *Byzantine failures* are failures when a node continues to run, but produces arbitrary incorrect results. An example of such a fault is that a node may produce arbitrary messages at arbitrary times. Pease, Shostak and Lamport [86] present the Byzantine General's Problem, where they created a fully fault tolerant system for these Byzantine failures.

2.1.3 Safety and cost

A system is in a **safe state** for a particular set of faults if these faults can impair the correctness of any critical services, but only in combination with some other faults [88].

Different approaches for robustness and fault tolerance can cause distinct kinds of costs in terms of money, time, effort, risk, computational resources, memory, etc. We consider the following classes of costs:

Development costs

Costs that occur in connection with system development, e.g., money spent for hardware and software purchases, licensing, development, and time-to-market.

Certification costs

Costs that occur in connection with system certification, e.g., efforts made in testing and verification.

Maintenance costs

Costs that occur in connection with system maintenance, e.g., money spent on replacement components and maintenance technicians, and system downtimes.

Operation costs

Costs that occur in relation to system operation, e.g., consumption of computational and memory resources, and costs of a critical failure.

2.1.4 Leader Election Strategies

Many applications and services for distributed systems are based on the existence of a separate process that coordinates the work of a set of processes. For example, LPEL has a coordinator node called *leader* that distributes tasks to the remaining nodes of the system. In this section, we describe the most common algorithms to perform leader election.

In the leader election strategies, it is necessary to detect the failure of the leader and to elect a new process to assume the role of leader. This election requires reaching an agreement among the processes as to which will be the new single leader. To determine the election of a process to become leader, processes need to have associated unique identifiers.

2.1.5 Rollback-Recovery Strategies

Rollback-recovery protocols attempt to recover the system from an inconsistent state to a consistent state. The definition of consistent and inconsistent state is found in the following subsection. One of the requirements of these protocols is to use stable storage that survives possible faults, which is used to restore the process from the stored data. The stable storage of each process can be local or global.

When a fault occurs, the data stored in stable storage are used to reset the process from a previous state before the failure has appeared. If the stored state is very recent, loss in computation is a minimal, otherwise it is not (Figure 2.3).

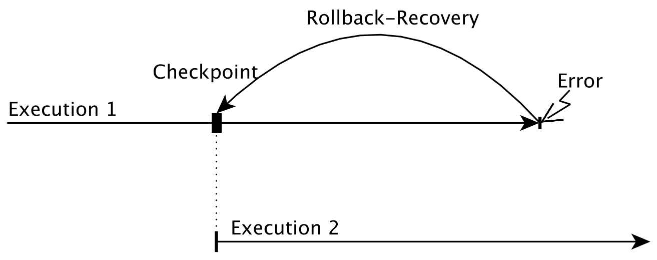


Figure 2.3. Example of rollback-recovery

The recovery information contains at least the last state stored of the participant processes, and these states are called **checkpoints**. Additionally, some rollback-recovery mechanisms need to store log events, such as interactions with input/output components, interprocess communication, etc. If the communication protocol is message passing, the rollback-recovery mechanisms are complicated because messages are events that lead to a certain dependence between processes.

Nowadays rollback-recovery protocols seek to be transparent (no human intervention) and to reduce the repair/recovery time. This transparency requires extra time from the runtime application because the system needs to store information and mechanisms to be automatic.

The interdependence of processes entails some problems: for example, if a process fails, it can mean that other processes that have not failed have to return to a previous state to synchronize with the failed process. This is called *rollback propagation.*

One point to worry about in rollback propagation is the **domino effect** [35, 111] during the restores. This phenomenon can appear in distributed systems in which each process takes its checkpoint independently. The need to establish a consistent state throughout the system can force other processes to restore a previous checkpoint, which may cause other processes to restore a previous checkpoint. This choice can continue until all the processes enter in the initial state because there are no more checkpoints to restore. Therefore, if checkpoints are not coordinated, a single failure can provoke a domino effect.

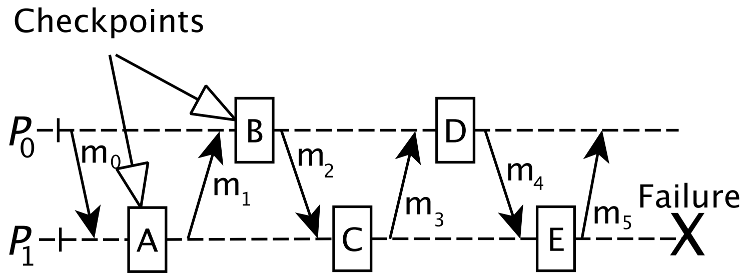


Figure 2.4. Example of the domino effect

The Figure 2.4 illustrates an example of the domino effect. If process *P1* fails, then *P1* restarts from checkpoint *E*. If message *m5* was sent after checkpoint *E*, then process *P0* needs to return to a state before in order to have no orphan message. Therefore, the problem now is in process *P1*, because the message *m4* is an orphan message. This problem continues until the processes find a state without orphan messages. In the worst case, this state can be the initial system state. The following lemma establishes the condition to avoid the domino effect.

Lemma 2.1.1 *If all the messages received by each process are eventually logged, there is no possibility of a domino effect in the system [46].*

The following rollback-recovery methods are defined in the literature [81, 49]:

- Coordinated checkpointing
- Uncoordinated checkpointing
- Communication-induced checkpointing

- Pessimistic logging
- Optimistic logging
- Causal logging

We grouped the checkpointing methods in the Checkpoint Subsection in section 2.1.5 and the logging methods in the Log-Based Subsection in section 2.1.5. These strategies have been studied [49] and compared according to the overhead there is over the protection and recovery step.

Description of processes

In distributed systems, a process can be informally understood as an executing program. Formally, a process is a unit of activity characterized by the execution of a sequence of instructions, a current state, and a set of associated system resources.

To understand what is a process and the difference between a program and a process, Andrew Stuart Tanenbaum [1] proposed an analogy: a computer scientist with a culinary mind bakes a birthday cake for her daughter; the scientist has the recipe for a birthday cake and a well-equipped kitchen with all the necessary ingredients: flour, eggs, sugar, milk, and so on. Placing each part of the analogy, it can be said that the recipe is the program (the algorithm), the computer scientist is the processor, and the ingredients are program entries. The process is the activity during which the computer scientist reads the recipe, obtains the ingredients and bakes the cake.

Therefore, distributed systems have processes, $P1$, $P2 \ldots PN$. These processes communicate with each other through messages. The aim of these systems is the cooperation of processes to execute a distributed application.

Then the execution of the process can be modeled as a sequence of deterministic state intervals, where each starts with the execution of a non-deterministic event. The events of one process can be non-deterministic or deterministic. **Non-deterministic** events are generated when the process receives a message from another process or when an internal event occurs inside a process. The remaining messages that are not included in the previous definition are **deterministic** for the process. Note that the messages sent by the process are deterministic. The idea of differentiating between deterministic and non-deterministic is called the *piecewise deterministic* (PWD) assumption. Then according to this assumption, all non-deterministic events that a process executes can be identified, and the information needed to reproduce these events can be logged.

The model is based on the dependencies between the states of the processes at some point when the processes are modified as a result of the communication between processes. Then the model helps us to study the correct functionality of the processes. All the processes are assumed to execute on fail-stop processors connected by a communication network, but the reliable delivery of messages

[1]Author of MINIX, a free Unix-like operating system for teaching purposes.

on the network is not required. The state of each process is represented by its dependencies, and the **global state** of a distributed system is represented by the collection of the process states and the states of the communication channels. Thus a consistent global state is one that may occur during a failure-free execution of a distributed computation. Consequently, a *consistent system* [14] state is one during which, if the state of the process reflects a message receipt, then the state of the corresponding sender reflects having sent that message. The aim of a rollback-recovery mechanism is to return the system from an inconsistent state because of failure to a consistent state.

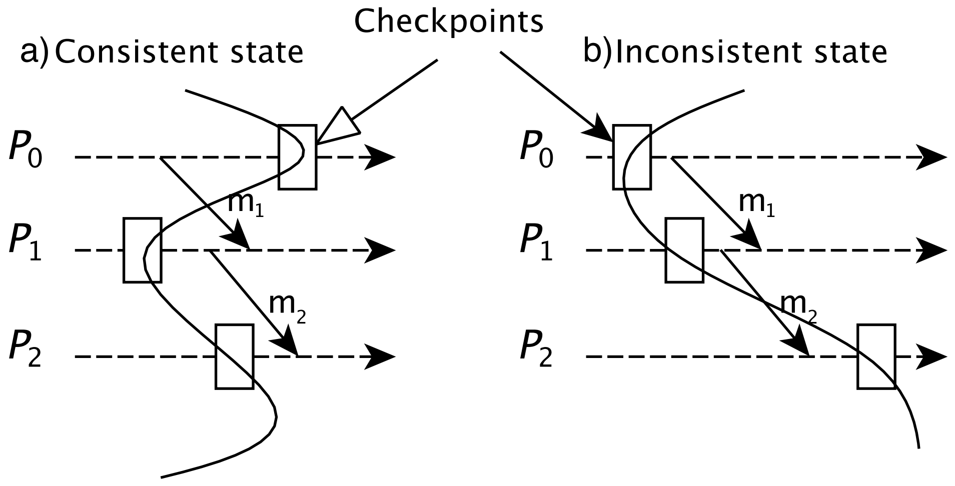


Figure 2.5. Example of consistent and inconsistent states

Figure 2.5 depicts two examples of global states. The first example is a consistent state because each received message has a corresponding message sent event. Each of these messages sent creates a non-deterministic event in the system. Process *P0* sends message *m1* to process *P1* and the process *P2* receives message *m2* from process *P1*. When any process fails, the system starts the restart mechanism from the last checkpoint in each process. Therefore, process *P0* knows that it sent message *m1*, but *P0* does not know if the message was received. After the restart, this message cannot be resent by *P0*. Consequently, *P1* does not receive *m1* after the restart. Accordingly, the system loses the information about *m1*. However, the system is consistent because all the processes have a record of the messages sent, and these processes know that all the received messages have a process that created this message.

In the inconsistent state, the process *P2* receives message *m2* and *P2* creates a checkpoint after the message has been received. Process *P1* sends the message without control. Then in the inconsistent state, process *P1* may fail after sending message *m2* to process *P2*. Afterwards the system uses the last checkpoint of each process to restart the state of the processes. Therefore, process *P1* can resend again message *m2* and create an inconsistent state in the system due to message duplication.

Process *p* becomes an orphan if *p* does not fail, but *p*'s state depends on a non-deterministic event e which cannot be recovered from the stable storage of process p. Process p is called on **orphan process** [73]. Another way to explain an orphan process is a process whose state depends on a non-deterministic event that cannot be reproduced during recovery. In the example figure the non-deterministic events are the messages sent and received from each process. A consistent state can have orphan processes: for example, in Figure 2.5(a), process *P1* is an orphan process because *P1* cannot recover message *m1* from process *P0*. Moreover, message *m1* is called an **orphan message**. Upon the recovery of all failed processes, log-based rollback-recovery guarantees the system does not contain any orphan process. Then log-based recovery satisfies the *No-Orphans Consistency condition*:

$$\forall e.\ \neg \mathit{Stable}(e) \implies \mathit{Depend}(e) \subseteq \mathit{Log}(e)$$

where *Stable(e)* is the function to store event e in the stable storage. This function returns true when the event is stored, otherwise it returns false. *Depend(e)* returns the group of all the processes affected by non-deterministic event e. *Log(e)* are all the processes that have stored event e in their stable storage or log file.

If any surviving process depends on an event e, then event e is logged in the stable storage, or the process has a copy of the determinant of event e. If neither condition is true, then the process is an orphan process because it depends on an event e that cannot be generated after the recovery. The reason for this is that the process creator of event e cannot recreate it once the restart mechanism is running.

Stable storage

In log-based rollback-recovery, checkpoints and event logs must be saved. This is why log-based checkpointing protocols use stable storage. Stable storage in log-based is only an abstraction, although it is often confused with the disk storage used to implement it. By definition, stable storage ensures that stored information is resistant to failures and their corresponding recoveries.

Then a stable storage is an abstraction of a perfect storage that survives processor or communication failures [41]. It provides atomic read and write operations; that is, these operations either execute completely, or do not execute at all, even when processor or communication failures occur. Stable storage is a basic requirement of a large number of fault-tolerance techniques, such as atomic actions [87], process checkpointing, and rollback recovery [48, 74, 82, 111]. Despite the importance of this abstraction, commercial systems do not provide a stable-storage service and, typically, it is left to the programmer to implement such functionalities whenever they are needed.

This requirement can lead to different kinds of stable storage depending on the failures tolerated:

1. If the system has to tolerate only one single failure, stable storage can be a volatile memory of another process. [7, 19]

2. If the system has to tolerate an arbitrary number of transient failures, stable storage can consist of a local disk.

3. If the system has to tolerate non-transient failures, stable storage must consist in a persistent medium outside the host on which the process running. An example of this kind of stable storage is a replicated file system.[29]

For this reason, we created the stable storages used in this thesis that it is like option two in the list before. Therefore each process has a local stable storage that can be read and written only for the process. So the each stable storage in the system is independent, the stable storage has a local synchronicity with the process.

Garbage Collection

In the log-based mechanism, checkpoints and logged messages are stored to recover the system in the event of failure. These consume storage resources. As processes collect information, a subset of stored information may become useless for recovery. Accordingly, the system needs a mechanism to remove useless information from processes. This mechanism is called garbage collection [76, 9].

Therefore, This information can be removed from stable storage only as long as it will not interfere with the system's ability to recover as and when needed. A common garbage collection mechanism aims to identify the recovery line and to remove useless information before that line.

Garbage collection is important for log-based protocols because removing the useless information from stable storage incurs overhead. Furthermore, log-based protocols differ from the information stored in stable storage and, therefore, differ in the complexity and invocation frequency of the used garbage collection.

Checkpoint

The checkpoint mechanism in distributed systems is complicated because each process has a parallel execution where there is no a global clock, which complicates the synchronization to create them [13]. In this case, both the frequency and content of the checkpoints are important features to define a checkpoint mechanism [78].

Frequency of checkpoints: checkpoint algorithms are run at the same time as the work of the process. Therefore, the overhead to create them must be minimized. Furthermore, checkpoints need to allow quick recovery without losing too much work done, which means creating them frequently is necessary. Hence the problem is if checkpoints are created often, overhead becomes very high. Yet if checkpoints are created rarely, when the restore method is called, the system loses a lot of previously done work and needs to do it again. Thus the frequency of the checkpoints is considered an important parameter in the checkpoint algorithm because the number of checkpoints needs to ensure minimal loss of information in the event of failure and to add a minimum overhead.

Content of checkpoints: The state of a process must be saved in stable storage in order to be used later for the recovery mechanism. The state includes a code, data variables and the content of memory, and registers that the process has been using when the checkpoint was created.

Depending on how the checkpoint is created, the following checkpoint mechanisms exit:

- Uncoordinated checkpointing
- Coordinated checkpointing
- Communication-induced checkpointing

Uncoordinated checkpointing

Using uncoordinated checkpointing has more advantages than coordinated and communication-induced checkpointing for distributed systems since there is no an agreement among processes to create checkpoints. Then each process can decide when its checkpoints are created. As a result, the main advantages are the autonomy of creating checkpoints and the scalability that these features provide.

Regarding autonomy, the recovery of a failed process can provoke an inconsistent state in other processes, and the other processes need to recover a previous state. Hence uncoordinated checkpointing cannot resolve the domino effect problem. Additionally, the checkpoints stored per process are difficult to handle per process and confer a high overhead during the execution time compared with the other checkpoint mechanisms [78].

During the system's failure-free execution, the processes calculate the interdependence among their checkpoints according to Bhargava's work [18]. This technique consists into attaching information about the checkpoint in each message sent by the process to another process. The receiver of the message uses the attached information to calculate the dependency between itself and the sender. So if a failure occurs in a process, the recovery mechanism initiates the rollback by requesting the information about the dependencies of messages related to the failed process to the system's remaining processes. When a process receives the request, the process stops its execution and replies with the information about the dependencies related with the failed process stored in its stable storage. The failed process waits until the relevant information is received from the other nodes. When all the processes reply, the failed process calculates the recovery line from itself and the processes with relevant dependencies. The failed process sends the result to the system's remaining processes. As a result, other processes may perform rollback to a previous checkpoint indicated by the failed process.

Coordinated checkpointing

Coordinated checkpointing strategies require all processes to come to an agreement and synchronise their checkpoints to create a global consistent state. This strategy confers simplicity to the recovery mechanism, and it is not affected by the domino effect because each process always restarts from the latest checkpoint stored in its stable storage. Furthermore, each process only needs to store the latest checkpoint because the other processes are self-sufficient to recover using the information stored in their stable storage. Consequently, the overhead used to store and manage previous checkpoints is removed in the coordinated checkpoint algorithms.

The main disadvantage of this method is the overhead to synchronize all the checkpoints created by the processes.
There are two ways to create a coordinated checkpointing: a blocking checkpoint or a non-blocking checkpoint. The blocking checkpoint [123] has a coordinator that forces processes to create checkpoints at a specific point in time. When processes receive the order from the coordinator, they stop their work and create a checkpoint all together. Regarding the non-blocking checkpoint [33], checkpoints are created at a specific time. All the processes have a clock. When the clock is ready, the processes do not receive more work, and they need to finish the work that they are presently doing. When all the processes are free, they start to create checkpoints all together. Afterwards, processes demand works to do. They wait until they are all free. However, one process may still be working and the remaining processes have to wait a long time without doing anything. The blocking checkpoint does not have this problem because, instead, the blocking checkpoint is more costly while executing the time of the each process's work than the non-blocking checkpoint.

Communication-induced checkpointing

Communication-induced checkpointing (CIC) [5] is balanced between uncoordinated and coordinated checkpointing. CIC allows processes to create checkpoints independently in order to avoid the domino effect. These checkpoints are called *local checkpoints.* Processes are also forced to create checkpoints, which guarantees the recovery line and the global consistent state of the system. These checkpoints are called *forced checkpoints.* These forced checkpoints are created when processes piggyback the received message using the information in the message to decide if the process needs to take a forced checkpoint or is not necessary. If a forced checkpoint is demanded, then the checkpoint must be created before the process reads the contents of the messages, which confers much latency and a high overhead to such methods. Another disadvantage of this method is the overhead in the messages sent during the communication between processes, because the information is related with forced checkpoints.

Log-based checkpointing approaches

After a failure occurs, the failed processes recover by using checkpoints and logged determinants to replay the corresponding non-deterministic events precisely since they occurred during pre-failure execution. While the process executes the rollback-recovery step, the other processes continue working and have no knowledge about the error in the affected process. It is desirable to minimize the amount of lost work by restoring the system to the most recent consistent global state, which is called the **recovery line**.

Log-based approaches avoid the domino effect since the failed process can be brought forward to the global state rather than individual processes being forced to rollback for consistency with the failed process.

These protocols allow each process to independently save its state avoiding spending additional time on the coordination between processes to save the process's state. The disadvantage of these protocols is the overhead that is added to save and store non-deterministic events in the stable storage. Some works have studied and compared the performance and overhead of log-based checkpointing protocols. [25, 26]

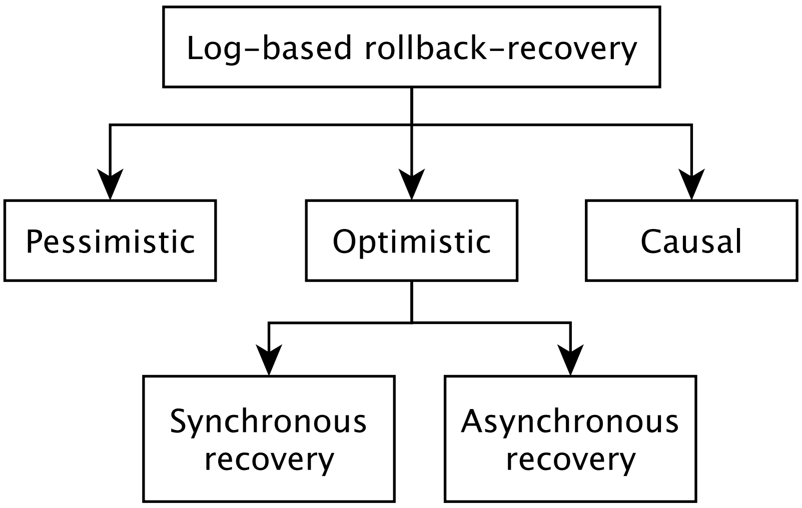


Figure 2.6. Classification of log-based rollback-recovery protocols

The Figure 2.6 illustrates the classification of log-based rollback-recovery protocols. These protocols offer the application a different performance, and have different recovery and garbage mechanisms. We now go on to explain these three log-based protocols and we add a figure to each one in order to better understand how they work.

Pessimistic log-based protocol

The pessimistic log-based protocol assumes that a failure can occur after any non-deterministic event during the computation. This situation is rare in a real world, which is why this is called pessimistic. In the pessimistic protocol, the message is always logged before it is processed. Therefore, a pessimistic protocol will not allow a process to send a message until all the delivered messages are in the log. The main idea is that the process stops whatever it is doing when it receives a message. Then the process logs the message into the log and resumes its execution. To place the message in the log file, the process must use a *synchronous logging* mechanism [126] to control file access. This synchronization mechanism is used to accomplished the always-no-orphans condition in the system:

$$\forall e.\ \neg Stable(e) \implies Depend(e) \subseteq Log(e)$$

This property states that if an event has not been stored in stable storage or a log file, then no process can depend on it. In addition to logging determinants, the process takes periodic checkpoints to minimize the amount of work that has to be repeated during recovery.

To recover a process from failure, the process rolls back to its latest checkpoint stored and plays back the messages stored in the log in the right order until the process enters in the pre-failure state. The rolling back of a process does not trigger the rollback of any other process thus, no Domino effect can occur. This method has three advantages:

1. The method allows processes to restart from their most recent checkpoint, which limits the extent of the execution that has to be replayed.
2. Recovery is simplified because the effects of a failure are confined only to the processes that fail.
3. Garbage collection is simple to implement.

The disadvantage is that synchronous logging incurs a high performance penalty during a failure-free operation.

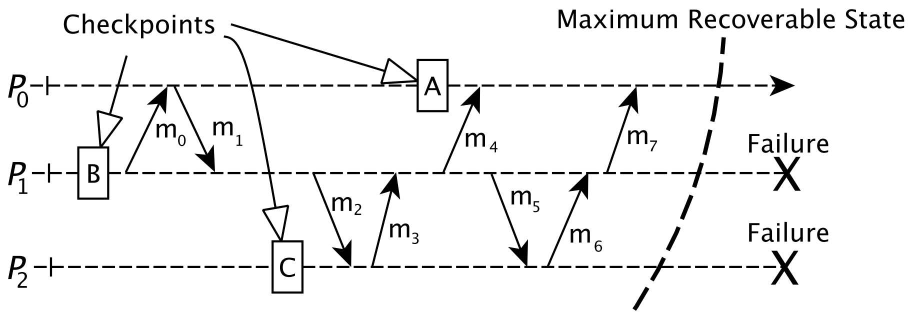


Figure 2.7. Pessimistic log-based protocol

Let's consider the example in Figure 2.7. During a failure-free operation, the logs of processes *P0*, *P1* and *P2* contain the determinants needed to replay the received messages. Then *P0* stores information about the *[m0,* *m4, m7]* messages, *P1* stores information about *[m1, m3, m6]* and *P2* saves information about *[m2, m5]*. Let's assume that processes *P1* and *P2* fail as shown, then the system restarts process *P1* from checkpoint *B*, and process *P2* from checkpoint *C*. After the restart, processes *P1* and *P2* use the information inside their logs to redeliver the same sequence of messages as during the pre-failure execution. This guarantees that *P1* and *P2* repeat exactly their pre-failure execution and resend the same messages. Hence once recovery is complete, both processes are consistent with the state of *P0*, which includes the receipt of message *m7* from *P1*. Until this point in the system, the three processes can be restored in case of a failure appears. Then, the three processes have a consistent recoverable state that, in case of failure, the three processes can be restored until that state. This state is represented as maximum recoverable state in Figure 2.7.

In a pessimistic log-based protocol, the status of each observed process is recoverable, and this property implies some advantages [49]:

- Processes can send messages to the outside world without using a special protocol.
- Processes can restart from the last checkpoint. Thus the overhead of computation is low.
- The recovery mechanism is simpler because the effect of failure affects only the process related with failure. All the other processes continue their execution. They are not orphan processes because the processes affected by failure restart to the consistence state, which includes the most recent interaction with other processes not related with failure.
- Garbage collection in pessimistic log-based protocol is simpler than optimistic and causal log-based protocol. Older checkpoints and non-deterministic events that occurred before the last checkpoint stored in stable storage can be removed because they are useless for the recovery mechanism.

Obviously the system needs to pay a price for these advantages. This price is deterioration in performance, which occurs from synchronous logging. For this reason, the implementations of this protocol must use special techniques to reduce the negative effects of such logging. The pessimistic logging (along with uncoordinated checkpoints) is used in the RADIC architecture [44].

Optimistic log-based protocol

The optimistic log-based protocol assumes that logging is always complete before a failure occurs. With this idea in mind, when messages are received, they are written in a volatile buffer, and are copied in stable storage or a log file asynchronously and at a suitable time. With this mechanism, the process execution is not disrupted, so the logging overhead is very low. It differs from the pessimistic protocol because in the optimistic protocol, processes asynchronously store information in the stable storage. [122] Thus the optimistic protocol has asynchronous logging.

Moreover, the optimistic protocol allows the temporary creation of orphan processes, but no orphan processes should exist by the time recovery is complete. When one process fails, the contents of its volatile buffer are lost, and the state intervals during which the process was started by such events cannot be recovered. If the process fails to send a message while the process is in any of these state intervals, the receiver of the sent message becomes an orphan process. To solve this problem, the process that receives the message needs to rollback. Therefore one process fails, which means that multiple processes have to do rollback as well. To avoid the domino effect, optimistic logging tracks the causal dependencies during failure-free execution. To be able to detect orphan processes, the dependencies between non-deterministic events need to be tracked during the entire execution, and dependency information must be piggybacked on the messages sent. Upon failure, dependencies are used to recover the last saved global state of the pre-failure execution during which the system had no orphan process. For this reason, the processes need to store multiple checkpoints.

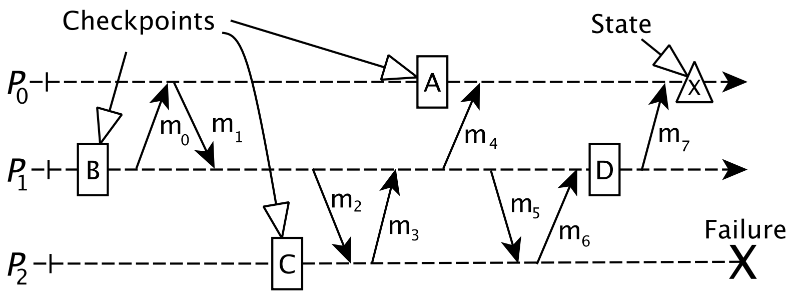


Figure 2.8. Optimistic log-based protocol

The Figure 2.8 shows that process *P2* fails before the determinant for message *m5* is logged to stable storage. Process *P1* then becomes an orphan process, so *P1* restarts from the last checkpoint *D* to try to undo the effect of receiving orphan message *m6*. Furthermore, the restart of *P1* forces process *P0* to restart in order to undo the effects of receiving orphan message *m7*. To perform these restarts correctly, optimistic log-based protocols track the messages between the processes during failure-free execution. Upon failure, this information is used to calculate and recover the latest global state of the pre-failure execution during which no process is an orphan process. However, non-determinant events are logged asynchronously. Therefore, the output committed in optimistic logging protocols requires a guarantee so that no failure scenario can revoke the output. For example, if process *P0* needs to commit the output in state

X, process *P0* must store messages *m4* and *m7* in stable storage, and ask process *P2* to store *m2* and *m5* because they are the messages related with *m1*. In this case, if any process fails, the system can restart to state *X*.

According to the above information, it is obvious that garbage collection mechanisms are not trivial for the optimistic protocol because this protocol needs to save the last created checkpoint. Therefore, to resolve the fail that affect process *P2* in Figure 2.8, process *P1* restarts from checkpoint *B* instead of from checkpoint *D*.

Recovery in optimistic protocols can be synchronous or asynchronous. In synchronous recovery[72], all the processes run the recovery mechanism to compute the most recent recoverable state based on both the dependencies of orphan messages and the information stored in stable storage. Then all the processes restart at the same time to the point calculated with the calculated information.

In asynchronous recovery mechanisms [122], the process with a failure sends a special message to inform other processes that it performs the restart. When a process receives this special message, the process restarts at a previous checkpoint if the process detects that it is an orphan process. As this process needs to restart, this process also sends the special message to inform all the other processes that it will perform the restart. With this mechanism, there can be multiple special messages from the same process. Then each process needs to control the dependencies between its state and the state of the process that sent the special message to detect if the process is an orphan process, and it needs to restart. All the restarts in this mechanism are asynchronous.

The advantage of this protocol is that the protocol incurs very little overhead during failure-free execution. Compared to the pessimistic log-based protocol, the disadvantages include the complication to create the recovery and garbage collection mechanisms. Moreover, the pessimistic log-based protocol needs only to keep the most recent checkpoint of each process, whereas the optimistic ones need to keep multiple checkpoints for each process. The output committed in an optimistic log-based protocol requires multihost coordination to ensure that a non failure scenario can revoke the output.

Causal Log-Based Protocol

It is a combination of the pessimistic log-based and optimistic log-based protocols used to create a protocol with the advantages of both. From the optimistic log-based protocol, the causal protocol obtains asynchronous access to store in the log file. Then the processes can read and write with asynchronous access, which means the access of the log file is not coordinated in time for the processes. From the pessimistic protocol, the causal protocol allows each process to commit output independently, it never creates orphan processes, and rollback is done with the most recent checkpoint. Thus the processes only have one checkpoint per process. In the causal log-based protocol, each process stores information about all the events that affect the process's state. This information is independent of other processes and allows the process to make its state recoverable by simply logging the available information. Therefore, the failure of one process only affects the failed process and the other processes do not rollback. Only the failed process rollsbacks to recover its pre-failure execution. As we have seen, the causal log-based protocol ensures the *always-no-orphans* property. For this purpose, the causal log-based protocol ensures that the determinants of each non-deterministic event that precedes the state of a process are either stable or available locally to that process.

For the recovery mechanism, each process stores its **antecedence graph**. This antecedence graph provides the history of all the non-deterministic events that have causal effect on the process state [49, 47]. In the antecedence graph, each node is a non-deterministic event that precedes the process's current state and the event occurs after storing the last checkpoint related to this process. The edges of the graph correspond to the relation between the events and help know the order of the creation of events. Any unnecessary nodes in an antecedence graph can be deleted when the new checkpoint is stored. Therefore, the antecedence graph of a process P is a directed graph *G(V,E)*, and:

- V is a set of non-deterministic events that precede P's current state (according to happened-before)
- E contains edge $v \to u$ if, and only if, v precedes u (according to happened-before)

The idea of the causal log-based protocol is when one process receives a message, this process starts to store an antecedence graph of all the messages related to the message received. Therefore, each message has attached an antecedence graph with information about the previous messages. Then as we show in Figure 2.9(a), *P0* receives message *m0*. First, *P0* starts to create an antecedence graph of all the messages created from message *m0*. At this moment, the antecedence graph has only one node with *m0*. Later, *P0* receives message *m4* with the antecedence graph attached. Then, *P0* checks the antecedence graph received in *m4* and sees that *m0* is in the antecedence graph. As a result, *P0* adds the new information to its antecedence graph, storing *m1*, *m2*, *m3* and *m4* to its stable storage or to its volatile log. Figure 2.9(b) is the result of this antecedence graph until this point. When a failure appears in the system, process *P0* can guide the recovery of processes *P1* and *P2* because *P0* knows the order of all the messages sent and received by *P1* and *P2* related to message *m0*.

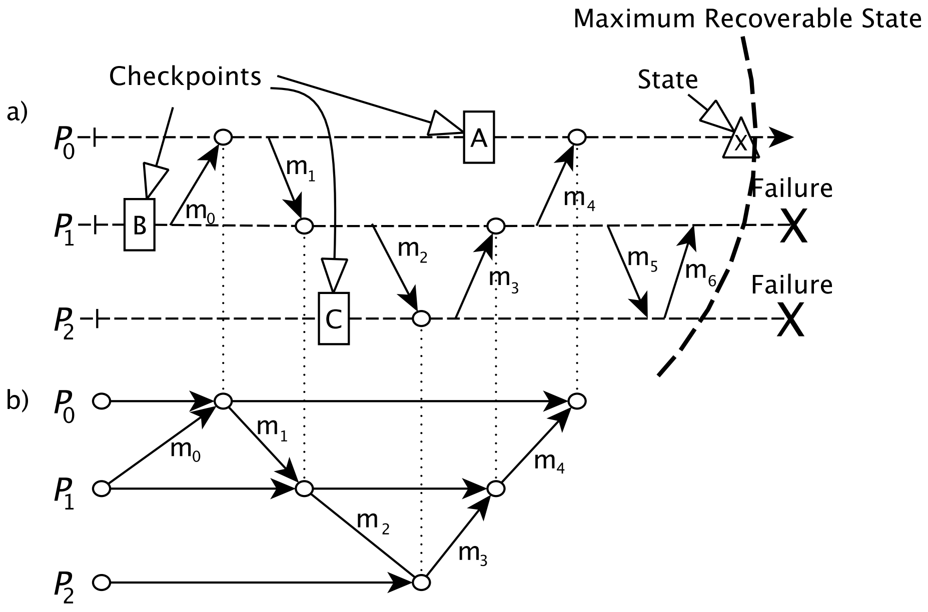


Figure 2.9. a) Causal log protocol. b) Antecedence graph of message *m4* **in** *P0*

It is important to note that messages *m5* and *m6* are not available in the antecedence graph of process *P0*. This is because each process creates an antecedence graph after the checkpoint is stored, and a message is received after the checkpoint has been created. In this example, the antecedence graph of message *m0* starts during process *P0* when *P0* receives *m0*. In the example, *P0* creates the antecedence graph of messages *m0* and *m4*, *P1* creates the antecedence graph of messages *m1* and *m3*, and *P2* creates the antecedence graph of message *m2*. In this example, processes *P1* and *P2* fail. Three processes (*P0*, *P1* and *P2*) work together to create a consistent state and to arrive at the maximum recoverable state.

In conclusion, causal log-based rollback-recovery protocols attempt to combine the advantages of low performance overhead and fast output commits, but need complex recovery and garbage collection. The information stored in stable storage is also more complex than in the other protocols.

Causal logging has the failure-free performance advantages of optimistic logging, but retains most of the advantages of pessimistic logging. The disadvantage of the causal protocol is the complexity of the recovery mechanism. Its advantages are that it:

- Avoids synchronous access to stable storage, except during the output committed.
- Allows each process to commit output independently. The process simply needs to save its log in stable storage
- Never creates orphans

- Limits the rollback of any failed process to the most recent checkpoint in stable storage.
- Reduces the storage overhead and the amount of work at risk.

2.2 Stream-Processing Networks

The idea of stream-processing networks is to create parallel networks in software and/or hardware. They are called **stream-processing networks** because the communications performed by the channels that pass information between modules are potentially infinite data sequences. These infinite data sequences are referred to as streams. A stream-processing network is a series of operations applied for each element in concurrent processing. Stream-processing is a computer-programming paradigm related to **SIMD** (Single Instruction, Multiple Data) that allows some applications to more easily exploit a limited form of concurrent processing. Such applications can use multiple computational units, such as FPUs on a GPU, without explicitly managing allocation, synchronization or communication among those units.

2.2.1 The S-Net Coordination Language

S-Net [58, 59, 106, 119] is a declarative coordination language used to describe streaming networks of asynchronous components at a high level of abstraction. [62] These components are interconnected by streams. The simplest components in S-Net are called **boxes**. Such a component has only one input and one output stream. Boxes communicate solely by means of the data received from the input stream and the data sent to the output stream.

A stream can contain zero, one data item or more, and boxes can receive and send these data items. Boxes can be deployed cheaply, and can be moved and replicated without giving rise to data integrity concerns.

Boxes are implemented in an appropriate programming language, e.g., ISO C or SAC [60]. Boxes are in fact **"black boxes"** from the implementation point of view. By being implemented in a different language, they do not expose any box internals to the S-Net level.

The streaming network in S-Net define asynchronous components inductively through algebraic formulas. There are only four essential composition patterns for **SISO** (Single-Input and Single-Output) components:

- Serial composition
- Parallel composition
- Serial replication
- Parallel replication

Each of these patterns has a corresponding *network combinator* in S-Net. These combinators create a new SISO component from one or two SISO components.

Generally, S-Net networks are asynchronous. One component sends output to the input buffer of the recipient component. The runtime system determines the size of these buffers, and the size of the buffer determines the degree of asynchrony between components. If we need to synchronize and combine the output of several components, there is a component in S-Net to do this, called **Synchrocell**. A synchrocell is the only stateful component type in S-Net. A synchrocell obtains several inputs and joins them into one output. The internal synchrocell state is made up of the records waiting to be synchronized.

Boxes send data items over streams to communicate with other boxes. These data items are organized as non-recursive tagged records with arbitrary non-record fields. The types of data associated with streams in an S-Net network are non-recursive, tagged variant record types. Like the function that actually implements a box, elementary types are indeed opaque to S-Net. Since all actual data are produced and consumed by box language programs, only the box language code can interpret data.

If there are more records field in the stream received by a box that the box can accept, then the excess fields are attached to the output stream of the box, and that part of the record is not processed by the box. This behaviour is called **flow inheritance** and it does not cause the program to fail. S-Net does not require explicit subtype declarations, but applies type inferences instead.

To better understand the S-Net language, we provide an example of a factorial function. The next code shows an implementation in ISO C. The ISO C function *factorial* only computes a single factorial number given suitable arguments.

Algorithm 1 Computing a single factorial number in ISO C.

```
function INT FACTORIAL($n$)
    int r, x;
    r = 1;
    x = n;
    while $x > 1$ do
        r = x * r;
        x = x -1;
    return (r);
```

Figure 2.10 provides a graphical representation of the network topology of the example that is the equivalent to the textual specification following the key word **connect** in Algorithm 2.

The S-Net network factorial shown in Algorithm 1 indeed transforms a stream of natural numbers into a stream of pairs, as reflected by its type signature $\{n\} \Longrightarrow \{n,fac\}$. The purpose of this example is to demonstrate the basics of the S-Net language. First we break down the problem into its atomic building blocks. The S-Net network uses five boxes for this example. The topology of

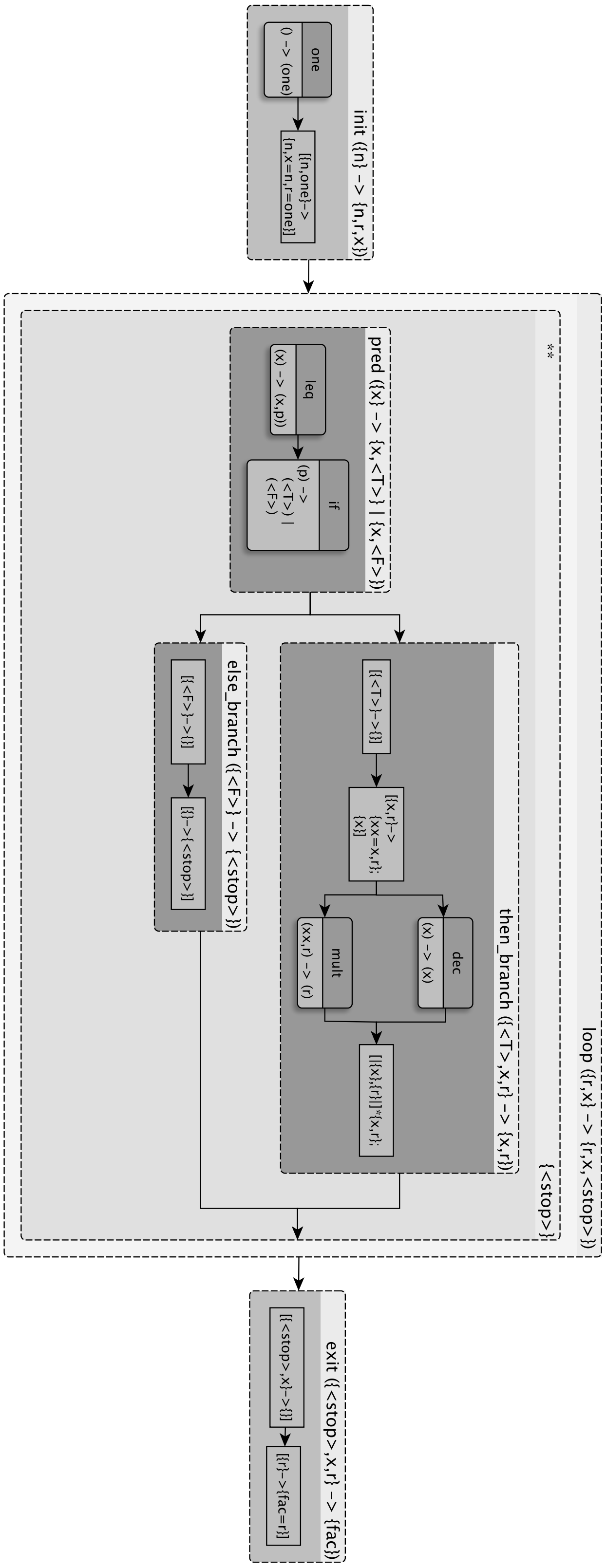


Figure 2.10. A graphical network representation of the factorial example.

the network factorial is fairly simple: a pipeline consisting of an initialization step, the main loop and a post-processing step.

The network *init*, very similar to the first few lines of the ISO C implementation, initializes new record fields *emphr* and *emphx* for the actual computation while the original argument *emphn* is preserved for global output. Whereas the renaming of one to *emphr* and the copying of *emphn* to *emphx* can be easily done at the S-Net level using a filter box, we employ a user-defined box to create a proper box language representation of the number one.

The while loop of the C function directly carries over to a deterministic star combinatory in S-Net. Note that the loop predicate (network *pred*) is entirely evaluated in the domain of a box language. The Boolean result is hidden in opaque record field *emphp* and can only be made accessible to S-Net by means of another box, if that takes field p, and it either yields a *tag T* or a *tag F* depending on its Boolean interpretation.

The network *then_branch* is like the loop body of the C function *factorial*. It starts by stripping off the *tag T* from each incoming record. Then it uses another filter box to duplicate each incoming record into one that is identical and one that only contains field x.

The best match rule of parallel composition combinatory plays a crucial role here in routing the $\{xx,r\}$ record to the box *mult* and the $\{x\}$ record to the box *dec*. Note that we need to rename field x as xx in order to circumvent the covariance restriction of parallel composition.

A subsequent synchrocell recombines records x and $\{r\}$ into a joint record $\{x,r\}$. Note that synchrocell is embedded in another serial replication. This combination of synchrocell and star combinatory is a very common design pattern in S-Net. It implements synchronization across an unbounded number of records: when $\{x\}$ arrives at the first synchrocell, the record is stored. If the next record is once $\{x\}$, a new synchrocell is created dynamically and the new synchrocell capture the $\{x\}$ record. The first synchrocell dies after synchronization, whose effect is that any subsequent records are directly sent to the second synchrocell instance.

Last but not least, the exit network strips off field x and tag stop from any record since they are only used internally by the factorial network. Eventually, field r, as it is used internally in factorial, is renamed as *fac* before a record leaves the whole network.

The sole purpose of our example is to illustrate the use of the various S-Net language features and their relationship to constructs known from conventional programming languages. Using boxes only for the most rudimentary computations and expressing anything else in S-Net are by no means representative of real-world S-Net applications. We expect boxes to represent substantial amounts of computational work and the S-Net layer to control only coarse-grained coordination aspects.

Algorithm 2 Computing a stream of factorial numbers in S-Net.

```
net factorial ({n} → {n,fac}) {
    box one (() → (one));
    box leq ((x) → (x,p));
    box if ((p) → (<T>) — (<F>));
    box dec ((x) → (x));
    box mult ((xx,r) → (r));

    net init ({n} → {n,r,x,}) {
    connect one .. [{n,one} → {n,x=n,r=one}];

    net loop ({r,x} → {r,x,<stop>}) {
        pred ({x} → {x,<T>} — {x,<F>})
        connect leq .. if;

        net then_branch ({<T>,x,r} → {x,r})
        connect [{<T>} → {} ]
            .. [{x,r} → {xx = x,r};{x}]
                .. (dec—mult)
                    .. [|{x},{r}|]*{x,r};

        else_branch ({<F>}) →{<stop>}
        connect [{<F>} → {}] .. [{} →{<stop>}];
    }
    connect (pred .. (then_branch || else_branch)) ** {<stop>});

    net exit ({<stop>,x,r} → {fac})
    connect [{<stop>,x} → {}] .. [{r} →{fac=r}]
}
connect init .. loop .. exit;
```

2.2.2 The S-Net Compiler

In order to manage the complexity of compiling a fully-fledged declarative S-Net code into a near machine-level representation, we define several intermediate variants of S-Net. A multistage compilation framework gradually transforms S-Net specifications into less abstract and less declarative codes.

Figure 2.11 (taken from the S-Net implementation report) shows a sketch of the overall S-Net compiler architecture. We define five compilation stages:

1. Preprocessing
2. Topology flattening
3. Type inference
4. Optimization
5. Post-processing

In addition, we have two auxiliary stages: **parsing** and **printing**. The five compilation stages share a common internal representation of networks. Auxiliary stages transform textual representations of networks into internal representations (parser), and vice versa (printer).

The compilation process may start and stop in any compilation stage. The exact stage in which to start is determined by an identifier in the source code. The first line of text must contain a special comment of the form:

//! snet code

If this identifier is not present, the compilation process starts at the very beginning. The final compilation stage is determined by a compiler flag. If that stage has been completed, the S-Net compiler prints the intermediate program representation to the standard output stream with the intermediate language identification properly set. The five intermediate languages are all variants of S-Net itself:

- $S\text{-}Net_{core}$
- $S\text{-}Net_{flat}$
- $S\text{-}Net_{typed}$
- $S\text{-}Net_{opt}$
- $S\text{-}Net_{final}$

First, an S-Net application is performed by the intermediate language S-Net $_{core}$. After that S-Net $_{flat}$ transforms the output of S-Net $_{core}$. In this way, a S-Net application go through these five intermediate languages before S-Net can performed. Therefore, the internal representation, the parser and the printer can be developed once, and parametrized for the different intermediate languages.

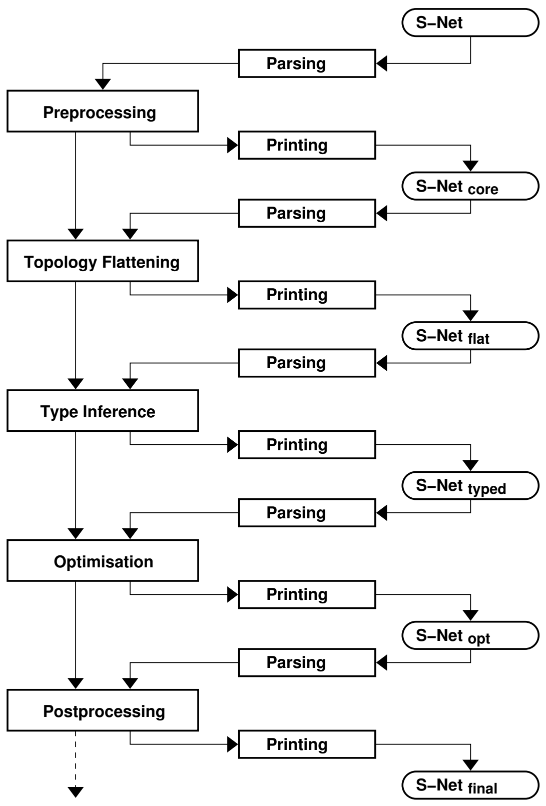


Figure 2.11. S-Net compiler architecture. [58]

The advantage of this multistage compiler architecture is that we can develop the individual parts mostly in isolation with well-defined interfaces in between them. Ease of use is still achieved by the compiler driver, which is responsible for user interaction and the orderly application of the individual compilation stages. The intermediate compiler phases can expect certain side conditions to hold apart from purely syntactical restrictions of the intermediate input language. In particular, the conditions that have been checked, enforced or created by preceding compiler phases do not need checking again. If for some reason they are violated, a compiler phase may arbitrarily fail in the attempt to compile the erroneous code. The feature of stopping and resuming the compilation process is exclusively intended for the sake of compiler development and testing. In a product version it is to be deactivated, which is good for robustness.

2.2.3 The S-Net Runtime Environment

A runtime environment implements the core behaviour of an S-Net language. The runtime system is a rich library of system calls for runtime representations of types and patterns, for setting up S-Net at runtime for the dynamic control of asynchronous S-Net components and the communication channels among them.

Figure 2.12 shows a sketch of the runtime system architecture. The common runtime interface is an abstraction layer that allows us to support different target

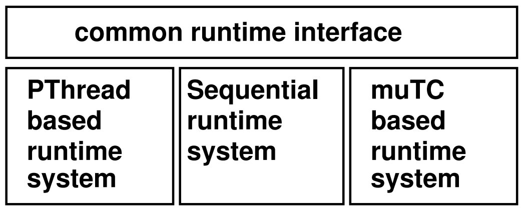

common runtime interface
PThread based runtime system
Sequential runtime system
muTC based runtime system

Figure 2.12. S-Net runtime system architecture. [106]

architectures without affecting the compilation and code generation process. For the time being, we envision three destination architectures:

- Sequential execution
- Multithreading based on PTHREADS and
- Multithreading based on μTC(an intermediate language for programming chip multiprocessors)

The common runtime interface shields the specific properties of these and other target architectures from the S-Net compiler and code generator. Actually, changing the target architecture does not even require the recompilation of an S-Net, but merely a link with a different runtime system implementation. Hence, the selection of a specific concrete target architecture forms part of S-Net deployment.

2.3 Data Flow Programming

Data flow programming is a paradigm based on the idea of a dataflow diagram. Applications are modelled as a directed graph or as networks similar to a dataflow diagram [31]. For this reason, applications are represented as a set of nodes that contain input and/or output ports in them. Each node can receive and send data. Nodes are connected by directed edges called *arcs* which define the flow of information between them. In contrast to the traditional control-flow model, a node is performed as soon as the node receives data from its input port.

2.4 The execution Layer LPEL

The Light-weight Parallel Execution Layer (LPEL) [109, 99] was designed to be used by S-Net and it provides an efficient and flexible execution platform for applications based on stream-processing for architectures with shared memory. A stream-based coordination program is a set of components connected by channels called **streams**. Additionally, S-Net uses LPEL to control the mapping and scheduling on shared memory platforms.

LPEL utilizes a user-level threading scheme for communication and threading protocols in user-space. For this reason, LPEL uses the services provided by the operating system and hardware, where LPEL runs to create threads, context switching in a user-space and atomic instructions.

With the LPEL architecture, it is possible to deliberate the allocation of available processing resources. LPEL also allows a large number of tasks to be handled at the same time with lock-free synchronization techniques and user-level threading.

The LPEL design is modular, and provides profiling information at execution time, as well as mechanisms for managing tasks in the user-space.

It is possible to use LPEL to know when the execution of a user-level thread is to start and stop. To know the execution time, LPEL can add a monitor for the stream, and a stream communication for monitoring is necessary to do this. This stream communication will be a slight overhead. It is also possible to switch off the monitoring. LPEL uses a monitoring framework to observe the mapping and scheduling activities. The following three kinds of monitor can detect particular events:

- **Mapping Event**: if the task is allocated to a worker, a mapping event occurs.
- **Scheduling Event**: if the task changes its state, a scheduling event occurs. There are five task states: task-created, task-blocked-by-input, task-blocked-by-output, task-resumed and task-destroyed.
- **Resource Load**: it is a monitor to see the overhead of workers.

LPEL provides good scalability and deployment of multiple available processor cores. Scalability is supported because LPEL employs lock-free data structures. Furthermore, LPEL's characteristics include:

- Support for non determinism by being able to test the availability of new data in input channels.
- Dynamic (de-)construction of the streaming network during runtime.
- Possibility of adapting the scheduling policy to the application requirements.

2.4.1 Architecture of LPEL

LPEL is implemented in the scheduling model of Parks [105] for process networks: The connection of the tasks is unidirectional and by streams, which are modeled as bounded buffers. Tasks are suspended from execution upon writing to a full stream and reading to an empty stream. This model enables easy implementation and renders parallel execution. Bounded buffers have the advantage of being able to provide back-pressure, but the downside is that they can lead to artificial deadlocks in circular networks. Tasks are not directly executed as operating system threads, but are executed as user-level threads in the

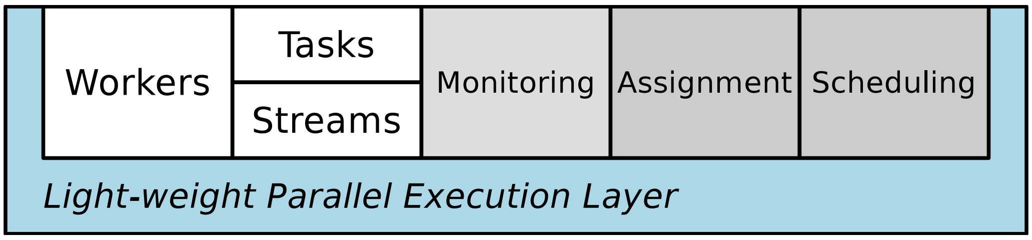


Figure 2.13. The architecture of LPEL [109]

LPEL context. Task management is tailored to LPEL task constructs and can be efficiently implemented.

Figure 2.13 presents the LPEL architecture and shows six components: workers, tasks, streams, monitoring, assignments and scheduling. We explain workers, tasks and streams, because monitoring, assignments and scheduling are not relevant for this thesis.

Workers and leader

The nodes in a stream-processing network are tasks that are executed in parallel execution threads in the system. **Workers** [99, 100] are responsible for executing tasks. Currently, LPEL defines one worker per processor. Thus each worker represents a CPU core or a hardware thread. Then the number of workers depends on the processing resources.

The reason for defining one worker per processor is to gain complete control over the scheduling in each core. At the beginning of LPEL program execution, one worker is defined as a **leader**.

Additionally, LPEL has a buffer that contains all the blocked tasks which is called **Central Task Queue** (CTQ). This buffer is managed only by the leader node of LPEL. Therefore, the leader distributes tasks from the CTQ to other workers by the mapping policy to decide which task is assigned to each worker using a time scheduling policy [100] to determine which task with a ready state is next dispatched.

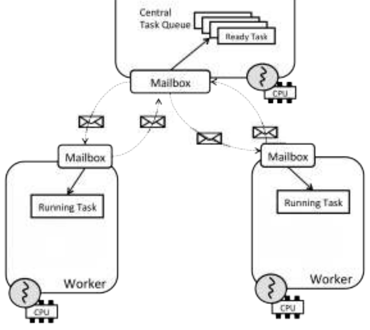


Figure 2.14. Example leader and two workers in LPEL

The task is returned to the leader when the worker completely performs the task or if the task is blocked. Afterwards the leader stores the task in the CTQ. An example of a LPEL system is shown in Figure 2.14.

Task

A program executed with LPEL is subdivided into different parts. For this reason, the S-Net boxes are mapped to tasks in LPEL. This subdivision is because each task can be executed in parallel. Each subdivisions is called a task. These tasks can be arbitrary chunks of work distributed into LPEL workers.

Tasks are not directly executed as operating system threads, but executed as user-level threads in the LPEL context. LPEL uses the streams to know if a task can be executed. The task reads the input messages from its input stream. If there is not enough information in the stream to start task execution, the task is *blocked* until the corresponding message arrives. The task is marked as *ready* and is executed by a worker when all the input messages needed to perform the task are in the input stream. Tasks are the processes that write the output messages in the output stream.

LPEL has a scheduling policy that determines if a task is in a ready state to be delivered to a worker. The state of a task is related to the availability of the input data in the input streams. This means that if a task wants to read from an empty stream, it is blocked and returned to the leader. If a task writes to an empty stream, another task can be unblocked when that stream is its input stream.

Mailbox

The communication in LPEL is by message passing. Then LPEL equips each worker and the leader with a mailbox to allow notification of a worker, either from outside of the worker or the leader. The message are stored to the receiver mailbox and the worker performs the message received from outside.

The mailbox basically consists of a message queue, in which messages are enqueued by other workers or the leader, and dequeued only by the owning worker or leader. The number of messages a worker can receive is not bounded.

In LPEL this messages can contain a task that it will be performed by the worker. Therefore, if the mailbox is empty and the worker is not running any task, then the worker is waiting for a message that contains a task from the leader.

Communication Model of LPEL: Streams

Streams are unidirectional communication channels among tasks, which is the only way that tasks can communicate with each other. Streams encapsulate the synchronization mechanism needed to block tasks and make them ready again when data become available. Therefore, if task T tries to write to a full stream or read from an empty stream, the task is *blocked*. Otherwise, task T changes its state to ready. The stream can be read and written at the same time by different workers on distinct processors. The idea is to use the task as a single producer and another task as a single consumer.

Streams are represented as a *First In, First Out* (FIFO) buffer for storing messages. These buffers are bounded to avoid memory overload. Bounded buffers have the advantage of being able to provide back pressure. Therefore, writing to a full stream causes the writing task to be blocked. The disadvantage is that streams can lead to artificial deadlocks in circular networks.

2.5 Chapter Summary

This chapter provided background information on fault tolerance including their features and classification. This chapter also described the execution model supporting S-Net stream languages and data flow programming. Furthermore, it is presented an instantiation of the execution layer LPEL that is used as benchmark platform for experiments to give fault tolerance to S-Net and LPEL.

Chapter 3

Related Work

The growing number of nodes in distributed systems has increased the probability of failures. Therefore, there is demand for development of techniques to detect and tolerate failures by losing a small fraction of computational performance. The idea of fault tolerance is to return the system after a fault happened in a consistent state so the system can recover execution from an inconsistent state.

Schoeder and Gibson [117] showed that failure rates in new hardware components of current systems increase compared with the old systems. Their work is based on the incidents that occurred over a nine year period in the Los Alamos Research Center. The conclusion of their work was that between 40% and 80% of failures, depending on the cluster size, are caused by a problem in a node of the cluster.

The study of fault tolerance techniques is not a new issue as they have been investigated for many years. Indeed they have become more important since the 1970s [34]. Additionally, evolution of computers and their application mean that these techniques are constantly studied in order to offer evolutionary and adaptive improvements to emerging systems.

In this thesis we focus on fault tolerance techniques related with leader election and rollback-recovery protocols for distributed systems.

3.1 Leader Election

Although the leader election problem has received plenty of attention in message-passing systems, very few solutions have been proposed for shared-memory systems. Guerraoui et al. present a leader election protocol for an synchronous shared memory [63]. Their approach is based on keeping an array of integer counters in a shared memory, with one entry per node. Each node has a static id, and the node with lowest id that is alive is assumed by all to be the leader. The approach is based on periodically polling shared memory by all the alive nodes in order to confirm the continuous availability of their currently assumed leader. The node that currently acts as leader periodically increases its counter in a shared memory to indicate its availability. The nodes that do not act as leader only read shared memory, but do not update it. For each node, the approach requires the periodic polling of all the node ids lower than its own. If a node sees that all the nodes with ids below its own have stopped to confirm their leader status, by increasing their counter value, than this node assumes the leader role, and consequently starts to periodically increase its own counter. This approach does not assume any knowledge about worst-case response times, but rather learns a sufficiently high timeout value over time by constantly doubling a delay factor. Once these timeouts have been established, the so-called *synchronous state* of shared memory is reached. The algorithm does not provide any guarantees about when this synchronous state is reached, and a node cannot be sure as to whether the current state is already a synchronous state or not.

In comparison to our leader election method, the strong side of the approach by Guerraoui et al.'s approach is that they do not rely on a synchronisation feature like CAS (there is more information about CAS in Section 5.1.1) as we do, but provide a more general application domain. In contrast, if CAS is available, our approach has several advantages. Guerraoui et al.'s approach relies on a node id used as a static priority to become leader, whose drawback is that the re-integration of a previously failed node causes unnecessary switches of the leader. Guerraoui et al.'s leader determination might sporadically lead to inconsistent views of the leader role should the current synchrony state not be stable. Furthermore, while we maintain only the data related to the current leader in the shared memory, Guerraoui et al. maintain a vector with entries for each node, and this makes their approach more data-intensive and more complex for updating purposes if new nodes arrive at the system.

A series of papers have focused on acquiring the best time and space complexity of leader election by Test-And-Set (TAS) implementations. Alistarh [3] implemented an algorithm, called RatRace protocol [4], that uses TAS to eliminate nodes, then the last node that is not removed is declared leader. The idea of this algorithm is to eliminate nodes with a sequence of sifting rounds, where each one reduces the number of survivors to roughly the square root of the number of processes that enter the round. More specifically, the algorithm is divided into $O(log\ log\ n)$ rounds. In each round, the node flips a binary biased coin. If the coin is 1, the node writes its identifier to the position of the round in the array in the shared memory, and then goes to the next round. If the coin is 0, the node reads the position of the round. If this array position has no data, the node goes to the next round. If there are data when the node reads the array from the shared memory, then nodes are marked as a losers, and lose the competition to become leader. When all rounds have finished, the survivor nodes participate in a RatRace protocol. In RatRace, each node will first acquire a temporary name, and then compete in a series of two nodes using TAS instances to decide the winner. The RatRace winner becomes the leader of the system.

The nodes in Alistarh's algorithm write to the shared memory in each round and the space complexity related to the shared memory is $O(n)$, where n is the number of nodes in the system. In our work, the space complexity is constant independently of the number of nodes. Another differences to our work is the step complexity. The step complexity in Alistarh's algorithm is $O(log\ log\ n)$, while our algorithm has $O(1)$.

Afek et al. [2, 127] extend the tournament binary tree idea of Peterson [107], where binary tree nodes are the winners of two processes that compete to be elected using TAS. When two nodes compete to become leader, the node that first modifies the shared memory is the winner and the other is the loser. Then the winner goes to the next leaf of the tree and competes against another node. If there are no more nodes to compete, the node that arrives at the last step of the binary tree becomes leader. The outcome of each node competition is stored in the shared memory, and the space complexity is $O(log\ n)$. The authors' algorithm has step complexity $O(log\ n)$ and was the first algorithm with logarithmic step complexity. Our solution has a constant step and space complexity.

Golab [56, 57] uses the Remote Memory References (RMRs) metrics to measure the performance of algorithms that solve a consensus and other related problems in two asynchronous shared memory models. The RMRs are defined to be the worst-case number of remote memory references by one process in order to enter and then exit its critical section. A *remote memory reference* is a shared variable access that requires an interconnect traversal. Golab provides deterministic Test-And-Set and Compare-And-Swap implementations with constant complexity in RMRs terms and an asynchronous shared memory model with no process failures. The algorithm proceeds in asynchronous merging phases, where nodes are divided into teams and a random node per team is elected as a leader of the team. As the leader of a team competes against another leader team. The winner team goes to the next step, where the leader is marked as a loser and a random node inside the team is selected as a leader. The loser team is marked as a loser. A separate flag register per node is stored in the shared memory to the losers and the winner.

In contrast, our approach uses only one variable in the shared memory, independently of the number of nodes. The algorithm does not need to pass steps/phases to elect a leader. In Golab's approach, only the leader of the team has access to the shared memory by TAS or CAS. In our case, all the nodes have access to the shared memory to find out who the leader is. However, Golab's approach is to better control how many times the node accesses the shared memory. Golab's algorithm cannot continue if a failure appears in a node that participates in leader election. In our case, we create a fault tolerance method that allows one or many failures at any time, and includes the reinsertion of crashed nodes into the system.

One characteristic of our algorithm is that although each node has a unique identifier, the system nodes do not need to know the identifiers of the other nodes as in the algorithms proposed in other works [51, 52].

3.2 Rollback-Recovery

Checkpointing and message logging protocols are well-known automatic and transparent fault tolerance techniques. In the literature, we can find three different classes of checkpointing protocols:

1. uncoordinated checkpointing
2. coordinated checkpointing
3. communication-induced checkpointing.

There are three different classes of log-based rollback-recovery, depending how and when events are logged:

1. Optimistic
2. Pessimistic
3. Causal [112]

.

Regarding log-based rollback-recovery mechanisms, they are also known as message-logging protocols and there are fault tolerance mechanism based on the Piecewise determinism assumption [122]: the rollback-recovery protocol can identify and store all the non-deterministic events created by each process. Each non-deterministic event contains all the necessary information to repeat the event in the recovery mechanism. Then if the PWD assumption holds, the restore mechanism of the log-based rollback-recovery protocol uses the non-deterministic event stored to recover the process before a failure appeared.

Some works about log-based rollback-recovery assume that non-deterministic events are only the events created by communication between processes, such as the reception and delivery of messages [46].

Section 2.1.5 offers a formal definition of checkpoint and log-based rollback-recovery protocols.

Johnson and Zwaenepoel [73] present a system that uses a pessimistic message logging protocol. Each sent message contains a unique identifier called a Send Sequence Number (SSN). This identifier is used to discard duplicate messages in the restore mechanism. Then a sender process asks the receiver for a Receive Sequence Number (RSN). A RSN is a counter of all the messages received by a process. Then the sender process delivers the received messages in accordance with the RSN assigned to each one.

Our proposal adds an identifier to each piece of data written or read in the stream, but the process cannot send messages between them. In our system, communication lies between the leader process and the worker process, and direct communication between different workers is not allowed.

MPICH-V [23] uses an uncoordinated checkpoint protocol along with log messages to preserve the status process and to recover a failed process if a fault appears. This implementation is designed for large-scale computing using heterogeneous networks, and it uses reliable stable memory called channel memories. Processes are created using MPI, and then each MPI process has access to a channel memory. When a process sends a message to another process, the message is sent to the channel memory of the receiver. Whenever the process wants to receive messages, the process reads its own memory channel. The idea is to store all messages and to use them to restore the process.

MPICH-V2 [27] is an improvement of MPICH-V, in which messages are stored locally by the sender and only information about the reception of the message is stored in the channel memory of the receiver.

If we compare MPICH-V and MPICH-V2 with our protocol, we use other non-deterministic events, and not only the events related to reading or writing a message (in our case, reading or writing to the stream). Additionally in our algorithm, the process stores a checkpoint of the task that is involved, and does not need to restart the process from the beginning, as in MPICH-V and MPICH-V2, because our approach processes messages independently.

Manetho [47] is the first implementation of a causal message-logging protocol. Manetho's idea was to force each process to keep an antecedence graph, which is a directed acyclic graph that records the causal relationship among all the messages sent in the system. Therefore, messages can be restored with the information in the antecedence graph. Thus when a process sends a message, it does not send all the antecedence graphs with the message, but only attaches the sufficient information to allow the receiver to reconstruct the antecedence graph with the received message. When a failure appears in a process, the other processes can use their portions of the antecedence graph to restore the failed process.

In our proposal, the restore mechanism is independent, which means that when a failure affects a process, this process can self-recover without help from any another process. Additionally, the processes in our algorithm do not have an antecedence graph.

Acharya and Badrinath [1] propose an uncoordinated checkpoint protocol for distributed applications in mobile computers, where the nodes are Mobile Hosts (MH). Then the system can arrive at a global consistent checkpoint without coordination messages. The protocol uses Mobile Support Stations (MSS) as stable storage to save MH's checkpoints. An MH takes a checkpoint, which is saved into the MSS when an MH connects to a new cluster and before disconnecting from another cluster. These authors define a message log that contains the state of an MH. With this approach they attempt to resolve the problems of search costs, frequent disconnections and lack of stable storage.

The Acharya's mechanism uses the last checkpoint stored into the MSS losing all the computation done between this checkpoint and the moment before the fault appears. In our case, we use log registers to restore the node's state closer to the previous moment of the fault appears, giving less computation overhead in the system.

Taesoon Park and H.Y. Yeom [104] present an optimistic message logging protocol for mobile computing systems. To resolve the stable storage problem in an MH, the MSS are responsible for performing the tasks of message logging and dependency tracking. Checkpoints are sent by the messages among MSS to reduce dependency, whereas the messages among the MHs that reside in the same MSS are traced from the message sequence. Given the little overhead and asynchrony, MHs are not involved in any coordination and the system can recover from failures using the information from MSS.

The Taesoon's approach can handle multiple and concurrent failures, exactly as our proposal can. The difference with our approach is that we are focused in distributed systems with shared memory instead of mobile computing systems. Additionally, Taesoon's approach restore the MH from the last checkpoint stored into MSS, causing that the checkpoint saved is old and the MH is restored to an old state losing computation time. Instead of that, we use log registers trying to restore the state of the worker to be closest to the last state of the processor before the fault appeared.

Guohong and Singhal [30] created a new approach called mutable checkpoints. The idea of mutable checkpoints is to modify the coordinated checkpoint protocol in an attempt to minimize the overhead generated by this protocol by taking checkpoints anytime and anywhere, i.e., local disk. So these mutable checkpoints help save space in the stable storage without forcing the transmission of large amounts of data over the whole network, thus cutting the time of overheads and the number of checkpoints taken. Therefore, the advantage of this proposal is the reduction in the number of irrelevant checkpoints, while the overhead of taking mutable checkpoints becomes negligible.

The Guohong's mechanism is a modification of the coordination checkpoint method and it can be affected by the domino effect, causing that the processors return to the initial state. Our approach is free of the domino effect, in case a fault affects one processor, not being disrupted the rest of processors for the apparition of this fault.

Chaoguang Men et al. [93] create a logging message protocol for cluster-based mobile ad-hoc wireless networks. Each cluster has a Cluster Head (CH) node where the checkpoint index, a node queue and a variable that counts the number of reply messages are stored. This CH is responsible for performing channel assignment, communication data and schedule intra-cluster traffic, whereas the node stores a checkpoint in its stable storage. Therefore, log messages are stored into the CH of the cluster. The CH sends special messages called beacon packets, to manage the nodes in the cluster. This special message is used to create two intervals to distinguish in which interval a node should take a checkpoint. The CH delivers all the parameters related to the node when a CH receives a checkpoint request from a node that attempts to restore its state.

The CH forces the nodes to take checkpoints, however, in our approach the checkpoint is taken for the nodes in an independent way, without being forced for other nodes or clocks. Then, we do not have the problem of a unique node that forces the rest of nodes to take checkpoints, being our nodes independent to take their own checkpoints and log messages without external demand. Additionally, our method does not use beacon packets to control the nodes, that implies the nodes have to be connected among them. Instead of using this beacon packets to restore the node, our approach is focused in an independent restore mechanism, that allows the nodes to work independently.

Sapna, Chen and Ying [55] describe a logging message protocol for a distributed system separated into clusters where checkpoints are taken if, and only if, a process exceeds a specific number of cluster changes (threshold). A process is forced to change its cluster by the system. The system reallocates the processes depending of the application performed by the cluster. Then a process takes a checkpoint if the process was forced to change its cluster a threshold number of times. This threshold is calculated with a performance model based on stochastic petri nets [16], where the parameters are the log arrival rate, failure rate and mobility rate of the process. As a result, each node has an independent checkpoint and message logging protocol, and the system optimizes the recovery cost, recovery time and storage issues.

The processes in Sapna's approach needs to exceed threshold to take a checkpoint. Sapna needs to constantly update the threshold depending of arrival rate, failure rate and mobility rate of the process into the cluster. Therefore, maintaining this threshold gives overhead into the cluster. Comparing with our approach, we do not define a threshold using the non-deterministic events to save the checkpoints and log messages necessary to restore the nodes in case of a fault appears. As a result, our approach does not need to use a process to calculate a threshold. Regarding the recovery cost, recovery time and storage issues, Sapna's approach and our approaches are optimized to work with lower system overhead, but the overhead in failure-free time is lower in our approach than Sapna's approach.

Jiyang et al. [71] propose a combination of communication-induced checkpointing and an optimistic message logging protocol for distributed systems, where the whole process shares stable storage. At the beginning, all processes use the communication-induced checkpoint protocol to create a checkpoint that accomplished consistent global state of the system and the processes store a copy of the checkpoint created into their local memory. After that, this protocol allows processes to take locally provisional checkpoints and to store them in stable storage whenever there is no contention for stable storage. So each process stores the provisional checkpoint in its local memory and then flushes it to stable storage when stable storage is ready. Regarding optimistic message logging, each process stores in its local memory all the messages sent and received after a provisional checkpoint has been created. After some time, all processes are forced to take a global state checkpoint and the log messages stored into their stable storage are removed when this new checkpoint is taken.

In our approach, we do not need to create a global consistent checkpoint to have a consistent global state, because we allow the processes to use the restore mechanism independently. Then, if a fault affects a process, the rest of the processes are not affected and can continue with their work without any notification of the fault occurrence. Additionally, the nodes in our approach do not use their local memory to store a consistent global state. As a result, our approach has lower storage complexity than Jiyang's approach.

Tong-Ying and Meng-Chang [77] propose a new independent checkpointing protocol for distributed systems organized as cluster-based structures. Each cluster has a coordinator process that manages all the communications inside the cluster and the garbage collection of stable storage. Then processes can communicate only with the coordinator in the same cluster as theirs. Each process has internal stable storage that can be read and written by the process and the coordinator process. A process periodically stores its current state, as well as a history of the messages received and sent in its stable storage. The author creates a rollback-recovery method using the information stored in the stable storage of each process. A processor is restarted from its most current checkpoint in the event of failure.

Instead of Tong-Ying's approach, each process in our mechanism has a stable storage that can be read and write only by the own process, and the coordinator (leader) cannot access to the stable storage. Other difference with our approach is the periodically storage of checkpoints. Then, Tong-Ying uses clocks to force the process to take a checkpoint after a specific time. In our case, we do not use clocks. Furthermore, the coordinator in Tong-Ying is responsible of the garbage collection, however, in our approach, the processes are responsible to remove the information inside their local stable storages.

3.3 Chapter Summary

In this chapter, we have presented a range of work related to two aspects of the context of this thesis. These are leader election and rollback-recovery. For leader election, we presented approaches where leader election methods use atomic operations to elect a new leader in the system. The second part of this chapter is focused on the approaches related to rollback-recovery mechanisms developed for distributed systems with stream processing networks.

Chapter 4

Fault tolerance for parallel stream-processing systems with shared memory

The main objective of fault tolerance techniques in distributed systems is the execution of work, even if a fault appears in any system component. These features are sought by paying a minimum overhead in the system's operational complexity.

This section presents the fault model by representing the faults that our system can handle. Afterwards, we describe various options for fault tolerance, and the advantages and disadvantages of each one, by starting with data replication and continuing with rollback-recovery strategies. At the end of this section, we can find a summary of the methods explained and why we selected the method used in this thesis.

4.1 Fault Model

From the fault tolerance perspective, we consider the computer system to be a subsystem that is embedded in its environment of the particular use case, which we call system. This environment embodied by the use case also determines what kind of faults we need to consider in order to make the overall system sufficiently robust. At the same time the environment also dictates what failure behaviour the computer system may exercise.

For faults, we consider the classification described in Section 2.1.1, where only the marked types (bold style) of the faults are considered in this thesis:

- Fault nature: **hardware(Digital and Analogical)**/software
- Fault duration: **permanent**/**transient**/Intermittent
- Fault extension: **local**/global
- Fault variability: **Determinate**/Indeterminate

We consider only the faults manifested at the hardware level because we assume that the faults manifested at the software level are primarily covered already within the software engineering and verification process. Obviously, errors may still remain in the software after verification, but we assume likelihood to be low enough to justify that decision. At the same time, we focus primarily on the faults whose states that do not change with time (determinate). Furthermore, we treat the faults that are permanent or transient in the system.

A characteristic feature of distributed systems that distinguishes them from single-machine systems is the notion of partial failure [85]: part of the system is failing while the remaining part continues to operate, and seemingly correctly. This property shows that distributed systems are affected by local faults. As a result, this thesis focuses on local faults.

Regarding the failures described in Section 2.1.2, we have to ensure that the affected node by fail-silent failure is rebooted and reinserted into the system when fail-silent is detected. Omission and timing failures are not dealt with in this thesis because our approach focuses on neither the communication protocol of the system, nor the synchronous distributed system. For Byzantine failures, this thesis is not focused to detect these kind of fault, instead of, the approach presented has a mechanism to handle and repair these failures when a Byzantine failures affects a data in the stream.

Regarding development, certification and maintenance costs, the created algorithms do not affect theses costs. The inclusion of fault tolerance and safety features in software designs typically incurs extra operational costs, and can reduce system performance. Operation costs are higher if the system employs the created algorithms compared to the system not using them. Losing performance is a small price to pay for acquiring a system that can support faults.

Additionally, the implementation of leader election and log-based rollback-recovery into LPEL and S-Net supports different flags to control that one that is desirable:

- ANY_FAULT_TOLERANCE: The user does not want any fault tolerance mechanism.
- LEAD_ELEC_FT_TYPE: Only the leader election mechanism is allowed.
- CHECK_RES_FT_TYPE: Only the log-based rollback-recovery is allowed.
- ALL_FAULT_TOLERANCE: The leader election and log-based rollback-recovery are used during the execution time.

If no flag is set, the application is executed as normal, but without the fault tolerance mechanism.

4.2 Fault Detection

In this thesis, we do not focused on fault detection. However, we present a simple timeout-based fault detector which is suitable if we can assume fail-silent behaviour of the failed nodes. The fail-silent detected in the experiments is based on the failure of the leader what causes the leader cannot continue working. Therefore, this fault detector, called notification message, is used by the leader election experiments and detects when the leader crashes.

We assume that all the messages are always sent and received without problems. Then workers cannot detect when the leader crashes. For this reason, we implemented the method called **notification message** to know if the leader is alive or not. Then the worker uses the notification message to detect the leader's failure and starts the leader election mechanism.

The timeout for this notification message is define as 3 seconds. This is enough time that a worker needs to receive the confirmation message from the leader in our benchmark examples.

Figure 4.1 shows the behaviour of the notification message method. There are two ways that a worker can detect the leader's failure.

1. When a worker returns a task to the leader, the worker awaits a specific timeout (3 seconds) for a notification message from the leader.

   If the worker does not receive the notification message from the leader within this timeout, the worker resends the task and again awaits the same specific timeout (3 seconds) for a notification message.

   In case that this second time the notification message is not received, the worker begins the leader election method.

2. When a worker is idle (non-busy) and is waiting for a new task from the leader. If the worker does not receive a task before an estimated timeout (3 seconds), the worker sends an alive message to the leader. This alive message is to know if the leader is still working or not. Therefore, if the worker does not receive the reply for the alive message sent from the leader before the estimated timeout ends, the worker assumes that the leader has crashed and the worker starts the leader election mechanism.

4.3 Leader Election Mechanism

The leader election is a fault tolerance mechanism that consists to elect a new leader after the leader of the system has crashed.

There are a lot of works related to a leader election mechanism for distributed systems. The problem of these works is that all nodes of the system need to participate in the leader election, causing that the system is stopped when the nodes are performing the leader election. In our case, we wanted a leader election mechanism that did not interrupt or force the nodes to wait for other nodes for electing a new leader. In addition, we wanted to have a fast and consistent leader election algorithm.

One of the thesis' goals is the creation of a fast and consistent leader election algorithm for S-Net and LPEL systems.

For this reason, we have created a leader election algorithm to elect a new leader among the non-busy nodes, while the busy nodes will update their information about the leader after finishing their task.

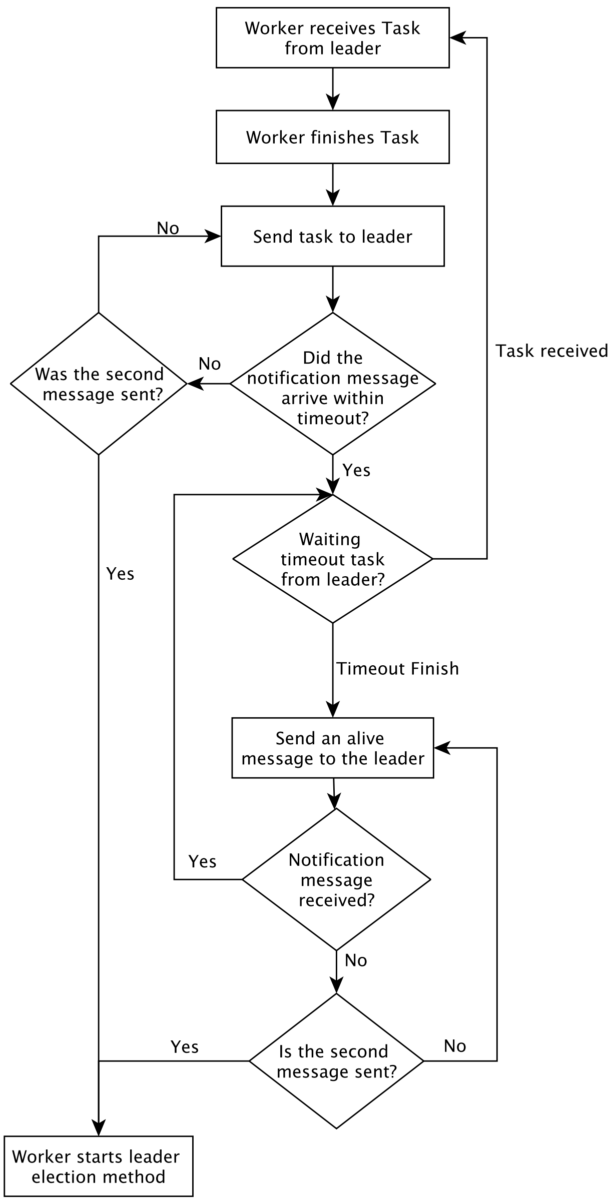

Worker receives Task from leader
Worker finishes Task
No
Send task to leader
Task received
No
Was the second message sent?
Did the notification message arrive within timeout?
Yes
Waiting timeout task from leader?
Yes
Timeout Finish
Send an alive message to the leader
Yes
Notification message received?
No
Yes
Is the second message sent?
No
Worker starts leader election method

Figure 4.1. Notification message behaviour of the workers

In our algorithm, nodes do not store the id of other nodes, not occupying the local memory of the nodes. Removing the necessity to know the id of other nodes, our algorithm allows to keep a low space complexity.

We also reduce the number of messages among nodes to 0, thanks to avoid the communication among nodes.

Regarding the consistency, the algorithm created assures that only one leader can be elected per leader election.

In case the leader is affected by a fault, it loses its condition of leader and it is reintegrated as a worker by the system.

In summary, our leader election has the following features:

- Nodes do not know the id of other nodes.
- The number of messages among nodes is 0, independent of the amount of nodes.
- Only non-busy nodes participate in the leader election.
- A crashed leader is reintegrated in the system as a worker.

With all these characteristics, our leader election achieves a low execution time overhead.

4.4 Rollback-Recovery Mechanism

One of the ideas of this thesis is to create a system that works for all the applications developed in S-Net and LPEL without using extra hardware resources. For this reason, we rule out replication mechanisms because a work is made up by two processes or more using hardware resources.

Regarding the rollback-recovery mechanism, Lemarinier's article [90] compares the coordinated checkpoint to the pessimistic log-based protocol implemented in MPICH-V [24]. The result of the paper shows better times for the pessimistic log-based protocol during the recovery mechanism than the coordinated checkpoint. Yet the latency and complexity of the recovery mechanism are bigger in the pessimistic log-based protocol. Based on this work, Paul's journal [118] indicates that it is important to choose a protocol that best suits the system, application and requirements used. For this reason, we create the Table 4.1, which shows the differences between the log-based rollback-recovery protocols in order to help us choose the best fault tolerance mechanism for S-Net and LPEL.

We seek a mechanism that avoids orphan processes and has low performance deterioration. So in uncoordinated checkpointing, CIC and optimistic logging are not selected for our approach. We intend to avoid interrupting the work of the process, thus we wish to create a non-blocking mechanism. As a result, coordinated checkpointing is also ruled out. Additionally, we seek a mechanism with no significant performance deterioration. Therefore, pessimistic logging is not suitable for our proposal. **Causal log-based rollback-recovery** is the fault tolerance mechanism used in this thesis.

The disadvantage of casual logging is the complexity of the recovery and garbage collection mechanisms. For the recovery mechanism, the algorithm of this thesis does not store an antecedence graph of each message. Instead we create a new role of a process called *Fault Tolerance Control*, which is used for processes to perform the recovery mechanism. For garbage collection, we explain in Section 6.1 a method to remove the useless data stored in each stable storage. With these modification, our log-based rollback-recovery mechanism can be used for distributed systems with stream networks, LPEL and S-Net.

	Uncoor. check-pointing	**Coor. check-pointing**	**CIC**	**Pessimistic Logging**	**Optimistic Logging**	**Causal Logging**
Checkpoint /process	Several	1	Several	1	Several	1
Domino Effect	Yes	No	Yes	No	Yes	No
Orphan processes	Yes	No	Yes	No	Possible (temporal)	No
Rollback extent	Unbounded	Last global checkpoint	Possibly several check-points	Last checkpoint	Possibly several checkpoint	Last checkpoint
Recovery data	Distributed	Distributed	Distributed	Distributed or local	Distributed or local	Distributed
Recovery protocol	Distributed	Distributed	Distributed	Local	Distributed	Distributed
Output commit-ted	Not possible	Global coordination required	Global coordination required	Local decision	Global coordination required	Local decision

Table 4.1. Summary of the log-based rollback-recovery protocol [49]

4.5 Chapter Summary

This chapter has presented an overview of the features developed and used for our leader election and log-based rollback-recovery mechanisms. The combination of our leader election and log-based rollback-recovery allows LPEL and S-Net to support permanent and transient faults that appear in the hardware of the system.

In spite of this thesis is not focused on the development of a fault detector, however, we created a notification message mechanism (timeout-based fault detector) used by our leader election experiments to detect when the leader fails.

We also explained why we decided to create a new leader election algorithm used by LPEL and S-Net, instead of using an existent one. We aimed to develop a fast and consistent leader election algorithm not presented in other algorithms, and we have achieved it.

With regard to the rollback-recovery mechanism, we chose the behaviour of the casual logging on account of the beneficial characteristics we exposed before in this chapter. Besides, we have partially modified its behaviour to adapt our algorithm for distributed systems with stream networks, such as, the creation of the FTC role and the absence of the antecedence graph. Its implementation is used by LPEL and S-Net.

Chapter 5

Leader Election

This section describes some foundations and assumptions required to understand our leader election method. First of all, our leader election method purely addresses the leader election problem after the failure of the leader, but does not address the issue of detecting the failure of the leader in the first place. The problem of leader failure detection is an orthogonal problem that is to be considered separately. The leader election method starts when the system detects the failure of the leader node. As only all the non busy nodes take part in the leader election method, no global consensus with all nodes is required in order to nominate the new leader. Our leader election method ensures that currently busy nodes will merely and correctly update their leader reference information.

5.1 Preliminaries

The failure of a node, or the unexpected termination of a task somewhere in the system, does not immediately affect the other nodes. Each node in the system can fail independently, which leaves the others still running and unaffected. Consequently, one of the advantages of using distributed systems is the independent failure of nodes [38]. As a result, there are only local faults in distributed systems, and global faults are avoided. Our algorithm has, therefore, focused on handling local faults.

Moreover, faults can appear in either software or hardware. Hardware failures used to be more common than software failures. Yet with all the recent innovations in hardware design and manufacturing, now they tend to be fewer and far apart, and most of these physical failures tend to be the related network or drive. This kind of faults can appear because of CPU errors or problems with energy, for example. This faults are shown in the thesis as a failures what causes the leader crashes and cannot continue working.

Any of these faults can be either fail-silent faults [28] (also known as fail-stops) or a Byzantine failure [32]. A fail-silent fault is one where the faulty unit stops functioning and produces no bad output. More precisely, it produces either no output or output that clearly indicates that the component has failed. A Byzantine fault is one where the faulty unit continues to run, but produces incorrect results. The faults treated in our election method are fail-silent hardware faults, because we are assuming that the leader cannot continue running after it is affected by a fault.

Although our leader election method runs even in the event of multiple failures in the system, this algorithm is only used for solving a leader crash and does not take other failures into account. If several nodes crash, including the leader, our algorithm utilizes the unaffected nodes. If all nodes except two fail at the same time, the leader election can elect one of the unaffected nodes as a leader node and the remain node as a worker. Normally, the systems need at least a worker and a leader. If there is not a leader in the system, the worker cannot receive a task that the worker will perform. On the other hand, if there is not workers in the system, the leader could not have worker to perform tasks.

Additionally, our solution is not restricted to any particular system topology, and the communication mechanism is based on messages passing among nodes. These nodes use the information in their local memory to communicate with the leader. If the leader has crashed, each node starts leader election, and either becomes a new leader or updates its local memory with the information in the shared memory.

5.1.1 Atomic operations

An atomic operation is defined as an indivisible operation with no intermediate state of its execution, and it cannot be interrupted or partially performed. The entire operation is either performed or not. This is important when multiple processes operate on the same memory area, and we need to guarantee that each node completely accesses. When a node runs an atomic operation, the other nodes see it as happening instantaneously. The advantage of atomic operations is that they are relatively quick compared to locks and do not suffer from deadlock and convoying. The disadvantage is that there is only a limited set of operations: read, write, and RMW (read-modify-write).

Depending on the platform used, we can find: Compare-And-Swap (CAS) or Load-Link/Store-Conditional (LL/SC).

The Compare-And-Swap (CAS) instruction was developed in the 1970s by IBM 370, and is now a standard instruction in many microprocessors, such as ARM microprocessors for mobile phones (ARMv6 and above). The CAS instruction is also used in several other architectures, such as x86, x86-64, or IA64 (Itanium).

The advantage of the CAS instruction is its atomicity. If multiple threads try to perform this operation simultaneously at the same memory address, only one thread succeeds and the others fail, but the other threads do not block. Instead they can continue with other operations or try again. This means that when multiple nodes attempt to update the same variable simultaneously using CAS, one wins and updates the variable's value, and the rest lose, but the losers are not punished by suspension [43].

In our leader election algorithm, we use a CAS instruction with three arguments:

- **First argument**: a memory reference.
- **Second argument**: data to be compared with the value in the memory referenced by the first argument.
- **Third argument**: if the comparison evaluates to true, the algorithm stores this value in the memory referenced by the first argument, otherwise it does nothing.
- **Return value**: the result of the comparison: *true* if the value in the memory referenced by the first argument equals to the second argument, *false* otherwise.

Algorithm 3 The instruction Compare-And-Swap (atomic)

```
1: function CAS(int $*p$, int $cmp$, int $new$)
2:     bool $retval := false$;
3:     begin_atomic;
4:     if $*p == cmp$ then
5:         $*p := new$;
6:         $retval := true$;
7:     end_atomic;
8:     return retval;
```

Algorithm 3 shows that if the current value of the variable at address *p* equals the value *cmp*, then the variable value referred to by *p* is set at the value of *new* and *true* is returned; otherwise *false* is returned and the variable value referred to by *p* does not change. All this is carried out atomically, indivisibly and with no partial effects.

If the Load-Link/Store-Conditional (LL/SC), the Load-Link (LL) instruction returns the current value of a memory location, the Store-Conditional (SC) instruction to the same memory location stores a new value, but only if no updates occurred at that location since the Load-Link instruction was performed. LL/SC is used on Alpha, PowerPC, MIPS and ARMv6 architectures, and can emulate CAS behaviour.

Algorithm 4 shows how CAS can be implemented using LL/SC. Implementing LL/SC in architectures that only support CAS primitive is possible, but not that straightforward. Details are provided in [94].

5.2 Algorithm

We now present our leader election method. We first start with a simplistic approach that can only tolerate persistent hardware faults (Algorithm 6). Then we extend this approach in order to also support non persistent hardware faults (Algorithm 7).

Algorithm 4 Implementation of Compare-And-Swap based on the LL/SC instructions

```
1: function LL(int *p)
2:     return *p;
3: function SC(int *p, int new)
4:     begin_atomic;
5:     if p has not been written to since this thread last called LL(p) then
6:         $*p = new$;
7:         return $true$;
8:     else
9:         return $false$;
10:    end_atomic;
11: function CAS(int *p, int old, int new)
12:    if $LL(p)! = old$ then
13:        return $SC(p, new)$;
14:    return $false$
```

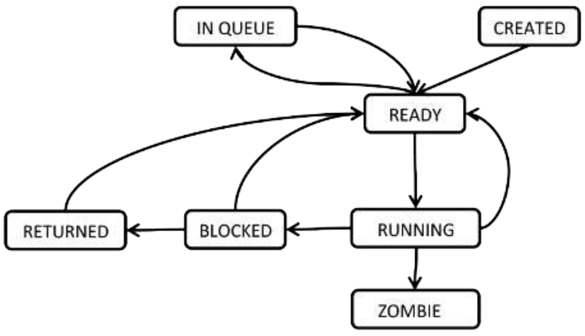


Figure 5.1. The different states of a worker.

At the beginning of our research, the leader election algorithm requires one variable in the shared memory: the id of the current leader. If the leader crashes, the remaining nodes compete to become the new leader. To compete during leader election, each node stores a copy of the current leader in its local memory. Figure 5.1 shows the behaviour of each node in the system to communicate with the leader.

Algorithm 5 shows a procedure called *CommunicateWithLeader*. This procedure is called per node when the node tries to send a message to the leader. The node uses the function *Communicate* (line 6) to send the message to the leader. This function is called using the information inside the node, such as the id to contact the leader (*nde.lid*) and the message sent to the leader (*buf*). At this point the node does not use the information stored in the shared memory, instead it uses the information stored in its local memory. If the message never reaches the leader, then the *Communicate* function returns *false*, otherwise, it returns *true*. Remember that the topic of this thesis is to repair the problem

Algorithm 5 Algorithm to send message to the leader

```
1: procedure CommunicateWithLeader(nde, buf)
2:     leaderContacted := $false$
3:     do
4:         // function Communicate(sender, message)
5:         // returns true if the message is received without problems
6:         if Communicate(nde.lid, buf) == $true$ then
7:             leaderContacted := $true$;
8:         else
9:             // communication with leader failed → volunteer to become new leader
10:            LeaderElection(nde);
11:    while leaderContacted == $false$;
```

when the leader stops work and not to detect if the leader crashes. For example, *Communicate* could use a handshake protocol with a timeout to be awaken. If the leader is broken, the node starts the leader election method.

After leader election, if the node does not become the new leader, it tries to communicate with the leader. At this time, the information stored in its local memory is updated in the leader election method.

Algorithm 6 Simple leader election algorithm (only for persistent faults)

```
1: procedure LeaderElection(nde)
2:     // Try to become new leader
3:     if CAS(shm.lid, nde.lid, nde.id) == $false$ then
4:         // Someone else already became new leader → store new leader id locally
5:         nde.lid := shm.lid;
6:     else
7:         // Only one node enters here to become new leader
8:         // Cleanup current task and change role to leader
9:         BecomeLeader(nde);
```

Algorithm 6 is the pseudocode for our leader election method. Nodes run this method after they realize the leader no longer works. The argument of this procedure is the information stored inside the node's memory. At the beginning of the method, a node compares the leader id in its local memory (*nde.lid*) with the identifier of the leader in the shared memory (*shm.lid*) using the CAS instruction. In Algorithm 6, the first CAS argument is the address of the leader id in the shared memory (*shm.lid*); the second CAS instruction parameter is the leader id contained in the node's local memory (*nde.lid*); the third one is the identifier of the caller (*nde.id*).

Two cases can occur with leader failure:

In the first case, when the node performs the *LeaderElection* method, the

Sequence	**Node 1**	**Node 2**	**Node 3**	**Shared Memory**
1	n_1.lid:3	n_2.lid:3	**LEADER CRASHES**	shm.lid:3
2		ContactLeader(n_2.lid:3)✗	Reinsertion(n_3.lid:-1)	shm.lid:3
3		$CAS(3,3,2)$✗	ContactLeader(n_3.lid:-1)✗	**shm.lid:2**
4		BecomeLeader()	$CAS(2,-1,3)$✗	shm.lid:2
5		**LEADER CRASHES**	**n_3.lid:2**	shm.lid:2
6	ContactLeader(n_1.lid:3)✗		ContactLeader(n_3.lid:2)✗	shm.lid:2
7			$CAS(2,2,3)$✓	**shm.lid:3**
8	$CAS(3,3,1)$✓		BecomeLeader()	**shm.lid:1**
9	BecomeLeader()			shm.lid:1

Table 5.1. The ABA problem

leader id stored in the shared memory differs from the leader id contained in the node. This means that this node loses the competition to become the new leader of the system. Accordingly, another node executes the CAS instruction, and then the leader id stored in the shared memory differs from the leader id contained in the loser node. Therefore, the node updates its local memory and, once updated, it finishes the method.

In the second case, the leader id stored in the node ($nde.lid$) and that stored in the shared memory ($shm.lid$) are equal. Then the CAS instruction sets the identifier of this node ($nde.id$) as the new leader id ($shm.lid$) in the shared memory. Then the node performs the function to modify its functionality to become the new leader of the system.

5.3 Reintegration of the nodes

According to our assumption, the algorithm allows the reintegration of leader or/and worker nodes after they crash. When the node is reintegrated into the system, the counter and the leader id stored in local memory are initialized to -1. Therefore, the reintegrated node tries to send a request task to the leader with the leader id equal to -1, and no node has an id that equals to -1. Thus when the node is reintegrated, it attempts to contact a non existent leader. As a result, the node starts the leader election and updates its local memory with the information stored in the shared memory.

5.4 Modification of the algorithm

The leader election algorithm that we presented in the previous section is affected by the ABA problem [114]. The **ABA problem** occurs when a node reads the shared memory twice: shared memory has the same value for both reads, and the value is used to indicate that nothing has changed. However, the other nodes executed between the two reads may change the variable, perform different work and then change the variable to its previous value. Then the first nodes believe that the shared memory value has not changed.

An example of this problem is shown in Table 5.1. This problem appears when node 1 is busy, while leader election is performed by other nodes and the leader changes.

Sequence	**Node 1**	**Node 2**	**Node 3**	**Shared Memory**
1	n_1.lid:3 n_1.cnt:1	n_2.lid:3 n_2.cnt:1	LEADER CRASHES	shm.lid:3 shm.cnt:1
2		ContactLeader(n_2.Lid:3)✗	Reinsertion(n_3.lid:-1, n_3.cnt:-1)	shm.lid:3 shm.cnt:1
3		$CAS_c(1,1,2)$✓	ContactLeader(n_3.lid:-1)✗	shm.lid:3 **shm.cnt:2**
4		$CAS_l(3,3,2)$✓	$CAS_c(2,-1,0)$✗	**shm.lid:2** shm.cnt:2
5		BecomeLeader()	**n_3.lid:2 n_3.cnt:2**	shm.lid:2 shm.cnt:2
6	ContactLeader(n_1.lid:3)✗	**LEADER CRASHES**	ContactLeader(n_3.lid:2)✗	shm.lid:2 shm.cnt:2
7			$CAS_c(2,2,3)$✓	shm.lid:2 **shm.cnt:3**
8	$CAS_c(3,1,2)$✗		$CAS_l(2,2,3)$✓	**shm.lid:3** shm.cnt:3
9	**n_1.lid:3 n_1.cnt:3**		BecomeLeader()	shm.lid:3 shm.cnt:3

Table 5.2. Solution to the ABA problem

As seen in the table, node 1 is busy when the leader (node 3) crashes. While node 1 is busy, node 2 becomes the leader and node 3 is reintegrated into the system and updates its local leader id with the leader id in the shared memory ($shm.lid = 2$). Afterwards, node 2 (current leader) crashes. Then node 1 finishes its task and tries to contact the leader in its local memory ($n_1.lid = 3$). However, node 3 is no longer the leader anymore, so node 1 starts the *LeaderElection* method. In parallel, node 3 participates in the leader election against node 1 and becomes the new leader of the system. But node 1 compares the local leader id stored in its local memory ($n_1.lid = 3$) to the leader id in the shared memory ($shm.lid = 3$), and modifies the information in the shared memory because node 1 thinks that node 3 is the leader when node 1 tries to contact node 3. As a result, the system has two leaders and these leaders can, in turn, have their own workers. Consequently, the system is split into two, and we wish to avoid this problem.

This situation is undesirable because the system is not prepared to control two leaders at the same time. So one leader and its workers do not perform any task and they are useless for the system. Additionally, another problem is the split of the system's resources.

We add a new variable called **counter** to indicate leader changes and to solve the ABA problem. Each node has a copy of this variable in its local memory and the shared memory also has a copy.

As a result, our improved algorithm uses two variables in the shared memory: the id of the current leader and a counter to avoid the ABA problem during the leader election. Each node also stores two variables: the copy of the id of the current leader and a copy of the counter in the shared memory. Figure 5.2 shows the behaviour of one node that executes the improved algorithm.

This counter is used to allow the reinsertion of the crashed leader into the system and to avoid the situation explained in Table 5.1 by solving the problem, as shown in Table 5.2, where CAS_c is the CAS instruction to compare the counter in the node's local memory to the counter in the shared memory; CAS_l is the CAS instruction to compare the leader id stored in the node's local memory to the leader id stored in the shared memory. This new variable affects only the leader election algorithm. Algorithm 7 shows the improved algorithm.

Algorithm 7 Modified leader election algorithm (to also support non persistent faults)

```
1: procedure LeaderElection(nde)
2:     // Try to become new leader
3:     if CAS(shm.cnt, nde.cnt, nde.cnt + 1) == false or CAS(shm.lid, nde.lid, nde.id) == false then
4:         // Someone else already became new leader → store new leader id locally
5:         nde.cnt := shm.cnt;
6:         nde.lid := shm.lid;
7:     else
8:         // Only one node enters here to become new leader
9:         BecomeLeader(nde); // Cleanup current task and change role to leader
```

Sequence	Node 1	Node 2	Shared Memory
1	n_1.lid:0 n_1.cnt:0	n_2.lid:0 n_2.cnt:0	shm.lid:0 shm.cnt:0
2	ContactLeader(n_1.lid:0)✗	ContactLeader(n_2.lid:0)✗	shm.lid:0 shm.cnt:1
3	$CAS_c(0,0,1)$✓		shm.lid:0 **shm.cnt:1**
4	$CAS_l(0,0,1)$✓		**shm.lid:1** shm.cnt:1
5	BecomeLeader()	$CAS_c(1,0,1)$✗	shm.lid:1 shm.cnt:1
6		**n_2.lid:1 n_2.cnt:1**	shm.lid:1 shm.cnt:1

Table 5.3. Example of two nodes competing to become the new leader

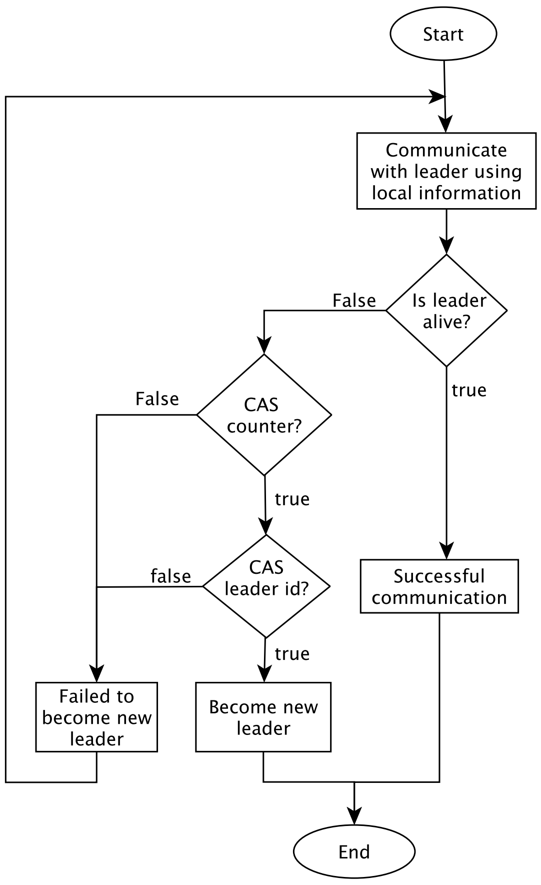


Figure 5.2. The possible behaviour of each node participating in the leader election algorithm

5.5 Examples of the Leader Election Algorithm

A few examples allow us to explain how our proposed leader election algorithm works in practice. Table 5.3 displays a typical situation in which two nodes compete in a leader election process. At the beginning, node 0 is the leader of the system. After the leader has crashed, nodes 1 and node 2 observe that the leader has crashed and start the leader election method.

Node 1 is faster than node 2 and first executes the CAS instruction to compare the counter in the shared memory ($shm.cnt = 0$) to the counter stored in its local memory ($n_1.cnt = 0$). Since they are equal, node 1 increases the counter in the shared memory by one ($shm.cnt = 1$). Afterwards node 1 uses another CAS instruction to compare the leader id in the shared memory ($shm.lid = 0$) to the leader id stored in its local memory ($n_1.lid = 0$). Since they are equal, node 1 sets its identifier, as a new leader, in the shared memory ($shm.lid = 1$). Then node 1 changes its role to the leader role in the system. Finally, node 1 finishes the leader election method.

Sequence	Node 1	Node 2	Shared Memory
1	n$_1$.lid:0 n$_1$.cnt:0	n$_2$.lid:0 n$_2$.cnt:0	shm.lid:0 shm.cnt:0
2	ContactLeader(n$_1$.lid:0)✗	ContactLeader(n$_2$.lid:0)✗	shm.lid:0 shm.cnt:0
3	$CAS_c(0,0,1)$✓		shm.lid:0 **shm.cnt:1**
4		$CAS_c(1,0,1)$✗	shm.lid:0 shm.cnt:1
5		**n$_2$.lid:0 n$_2$.cnt:1**	shm.lid:0 shm.cnt:1
6	$CAS_l(0,0,1)$✓		**shm.lid:1** shm.cnt:1
7	BecomeLeader()		shm.lid:1 shm.cnt:1
8		ContactLeader(n$_2$.lid:0)✗	shm.lid:1 shm.cnt:1
9		$CAS_c(1,1,2)$✓	shm.lid:1 **shm.cnt:2**
10		$CAS_l(1,0,2)$✗	shm.lid:1 shm.cnt:2
11		**n$_2$.lid:1 n$_2$.cnt:2**	shm.lid:1 shm.cnt:2

Table 5.4. One node updates before the new leader is elected

Sequence	Node 1	Node 2	Node 3	Shared Memory
1	n$_1$.lid:0 n$_1$.cnt:0	n$_2$.lid:0 n$_2$.cnt:0	n$_3$.lid:0 n$_3$.cnt:0	shm.lid:0 shm.cnt:0
2	ContactLeader(n$_1$.lid:0)✗	ContactLeader(n$_2$.lid:0)✗	ContactLeader(n$_3$.lid:0)✗	shm.lid:0 shm.cnt:0
3	$CAS_c(0,0,1)$✓			shm.lid:0 **shm.cnt:1**
4		$CAS_c(1,0,1)$✗		shm.lid:0 shm.cnt:1
5		**n$_2$.lid:0 n$_2$.cnt:1**	$CAS_c(1,0,1)$✗	shm.lid:0 shm.cnt:1
6			**n$_3$.lid:0 n$_3$.cnt:1**	shm.lid:0 shm.cnt:1
7	$CAS_l(0,0,1)$✓	ContactLeader(n$_2$.lid:0)✗	ContactLeader(n$_3$.lid:0)✗	**shm.lid:1** shm.cnt:1
8	BecomeLeader()	$CAS_c(1,1,2)$✓		shm.lid:1 **shm.cnt:2**
9		$CAS_l(1,0,2)$✗	$CAS_c(2,1,2)$✗	shm.lid:1 shm.cnt:2
10		**n$_2$.lid:1 n$_2$.cnt:2**	**n$_3$.lid:1 n$_3$.cnt:2**	shm.lid:1 shm.cnt:2

Table 5.5. Two nodes update before the leader is elected

While node 1 changes its worker role to leader role, node 2 compares the counter in the shared memory ($shm.cnt = 1$) to the counter in the local memory ($n_2.cnt = 0$) using the CAS instruction. In this case, the data that the CAS instruction compares are not equal. As a result, node 2 does not modify the shared memory in this leader election and updates its local variables with the information in the shared memory that node 1 previously set as a new leader.

Table 5.4 shows an example of two nodes, where one node updates its local information before another node modifies the leader id. As a result, node 2 tries to contact the incorrect leader ($n_2.lid = 0$) and starts the leader election method. Node 2 has the same counter value ($n_2.cnt = 1$) as the shared memory ($shm.cnt = 1$). So node 2 increases the counter in the shared memory ($shm.cnt = 2$). Node 2 tries using CAS to modify the leader id in the shared memory, but does not have the same leader id as the same memory. Therefore, node 2 fails when using CAS and updates its local memory. In Table 5.5, node 3 does not increase the counter in the shared memory. Therefore, the counter is always increased by the node that becomes leader, and can perhaps be increased only once by another leader election participant.

Table 5.6 shows an example in which node 1 crashes after node 1 modifies the counter. Afterwards, node 1 is reinserted into the system. The counter and the leader id in the local memory of node 1 are initialized to -1 ($n_1.lid = -1$ and $n_1.cnt = -1$). So node 1 is reinserted as a worker into the system.

Table 5.7 displays an example in which node 1 modifies the shared memory

Sequence	Node 1	Node 2	Shared Memory
1	n_1.lid:0 n_1.cnt:0	n_2.lid:0 n_2.cnt:0	shm.lid:0 shm.cnt:0
2	ContactLeader(n_1.lid:0)✗	ContactLeader(n_2.lid:0)✗	shm.lid:0 shm.cnt:1
3	$CAS_c(0,0,1)$✓		shm.lid:0 **shm.cnt:1**
4	✗	$CAS_c(1,0,1)$✗	shm.lid:0 shm.cnt:1
5		**n_2.lid:0 n_2.cnt:1**	shm.lid:0 shm.cnt:1
6	Reinsertion(n_1.lid:-1, n_1.cnt:-1)	ContactLeader(n_2.lid:0)✗	shm.lid:0 shm.cnt:1
7	ContactLeader(n_1.lid:-1)✗	$CAS_c(1,1,2)$✓	shm.lid:0 **shm.cnt:2**
8	$CAS_c(2,-1,0)$✗	$CAS_l(0,0,2)$✓	**shm.lid:2** shm.cnt:2
9	n_1.lid:2 n_1.cnt:2	BecomeLeader()	shm.lid:2 shm.cnt:2

Table 5.6. Node 1 crashes after it modifies the counter in the shared memory

Sequence	Node 1	Node 2	Shared Memory
1	n_1.lid:0 n_1.cnt:0	n_2.lid:0 n_2.cnt:0	shm.lid:0 shm.cnt:0
2	ContactLeader(n_1.lid:0)✗	ContactLeader(n_2.lid:0)✗	shm.lid:0 shm.cnt:1
3	$CAS_c(0,0,1)$✓		shm.lid:0 **shm.cnt:1**
4	$CAS_l(0,0,1)$✓	$CAS_c(1,0,1)$✗	**shm.lid:1** shm.cnt:1
5	✗	**n_2.lid:1 n_2.cnt:1**	shm.lid:1 shm.cnt:1
6	Reinsertion(n_1.lid:-1,n_1.cnt:-1)		shm.lid:1 shm.cnt:1
7	ContactLeader(n_1.lid:-1)✗		shm.lid:1 shm.cnt:1
8	$CAS_c(1,-1,0)$✗		shm.lid:1 shm.cnt:1
9	**n_1.lid:1 n_1.cnt:1**		shm.lid:1 shm.cnt:1
10	ContactLeader(n_1.lid:1)✗	ContactLeader(n_2.lid:1)✗	shm.lid:1 shm.cnt:1
11	$CAS_c(1,1,2)$✓		shm.lid:1 **shm.cnt:2**
12	$CAS_l(1,1,1)$✓	$CAS_c(2,1,2)$✗	**shm.lid:1** shm.cnt:2
13	BecomeLeader()	**n_2.lid:1 n_2.cnt:2**	shm.lid:1 shm.cnt:2

Table 5.7. Node 1 crashes after modifying the leader in the shared memory and before taking the leader role

and crashes before taking the leader role.

5.6 Evaluation

The evaluation of the leader election mechanism in sequential algorithms is measured using time and space complexity. These measures are represented in terms of the *lower bounds* that represent the best case, and the *upper bounds* represent the worst case [20]. The performance of time and space complexity for distributed systems is slightly different and message complexity is also used to evaluate algorithms. Therefore, the following complexity measures are used in this thesis:

1. **Space complexity per node**: the memory required by each node for the best, average and worst cases.

2. **Time complexity per node**: measures the processing time per node.

3. **System-wide time complexity**: measures the total processing time of the leader election for all the nodes.

Additionally in our evaluations, we use the standard **big 'O' notation**, also called **Landau's symbol** [21]. The big 'O' notation is a standard used to describe the performance or complexity of an algorithm. The big 'O' notation seeks to describe the relative complexity of an algorithm by lowering the growth rate to the key factors when the key factor tends toward infinity. As a result, the big 'O' notation has the following rule for multiplication by a constant. Let k be a constant, then it follows:

$$O(kg) = O(g) \qquad | \text{ if } k \text{ is non zero}$$

5.6.1 Space complexity per node

The space complexity per node is the amount of space memory required for a node to solve the algorithm. In this case, the information stored in each node is composed of an integer with the leader id, and another integer that counts how many times leader election has been done. The integer type is a primitive data type in computer science. Therefore when using the Big 'O' notation to represent space complexity, the space complexity per node is $O(1)$.

5.6.2 Time complexity per node

The time complexity per node is calculated using the total *number of steps* that a node needs in the worst case to perform the leader election algorithm. Each step is an operation that takes a fixed amount of time to perform. The execution of an algorithm may halt from time to time because of the interruptions generated by the operating system. However, the time complexity measure is immune to these unpredictable interruptions.

Table 5.8 shows that the time complexity of Algorithm 5 is $O(1+(2+1+1)*x)$, where x is the time in which the *doWhile* loop is performed. Lemma 7.1.6 shows that the maximum times of one node that performs the *doWhile* loop is 2. Therefore, the time complexity per node at Algorithm 5 is $O(1)$.

Instruction	**Time complexity**
leaderContacted := false	1
If (Communicate(node.lid, buf) == true)	2
leaderContacted := true;	1
LeaderElection(node);	1
leaderContacted == false	1
doWhile()	x
Worst case	1 + (2 + 1 + 1) * x

Table 5.8. Time complexity of the algorithm to send messages to the leader

Regarding Algorithm 7, time complexity focuses on the time used to read/write to the shared memory because these actions use more time than other instructions. Table 5.9 shows the instructions of Algorithm 7 that have access to the shared memory. If the CAS instruction returns *true*, this means that the node makes one read and one write to the shared memory. The node only makes one read to the shared memory if the CAS instruction returns *false*.

In the worst case, a node succeeds with the CAS instruction related to the counter and fails with the CAS instruction related to the leader id. Then the node updates its local information with two reads to the shared memory. As a result, and in the worst case, a node reads four times the shared memory and writes one time to the shared memory. Therefore, the time complexity per node in Algorithm 7 is $O(4 * T_{\text{read}} + T_{\text{write}})$, where T_{read} is the time needed by the node to read the data from the shared memory; T_{write} is the time needed by the node to write data to the shared memory.

Instruction	**Read**	**Write**
CAS(shm.cnt, node.cnt, node.cnt + 1)	1	1
CAS(shm.lid, node.lid, node.id)	1	0
node.lid := shm.lid	1	0
node.cnt := shm.cnt	1	0
Worst case	4	1

Table 5.9. Accesses to the shared memory by one node in the algorithm to elect a leader

5.6.3 System-wide time complexity

If the processing in the distributed system occurs at all the processors concurrently, then the system time complexity is not n times the time complexity per node. However, if the executions by the different processes are done serially, as with the case of an algorithm in which only the unique token-holder is allowed to execute, then the overall time complexity is additive. Therefore in this work, the nodes perform the algorithm in parallel and they can take different behaviours in the election method as Figure 5.2 shows.

As we stated in the previous section, the time complexity for Algorithm 7 is based on the reading and writing instructions to the shared memory because these instruction are more expensive in time complexity than the other algorithm instructions. We present the time complexity for one node in the worst case.

The nodes have three different behaviours for time complexity depending on the way they acquire them in Figure 5.2:

1. The node becomes leader
2. The node fails in the CAS instruction when comparing counters
3. The node fails in the CAS instruction when comparing leader ids

In the first case, the node wins the leader election, and reading and updating two times the shared memory using the CAS instruction. Thus the time complexity used for a winner node is:

Sequence	**Node 1**	**Node 2**	**Node 3**	**Node 4**	**Shared Memory**
1	n_1.lid:0 n_1.cnt:0	n_2.lid:0 n_2.cnt:0	n_3.lid:0 n_3.cnt:0	n_4.lid:0 n_4.cnt:0	shm.lid:0 shm.cnt:0
2	ContactLeader(n_1.lid:0)✗	ContactLeader(n_2.lid:0)✗	ContactLeader(n_3.lid:0)✗	ContactLeader(n_4.lid:0)✗	shm.lid:0 shm.cnt:0
3	$CAS_c(0,0,1)$✓				shm.lid:0 **shm.cnt:1**
4		$CAS_c(1,0,1)$✗			shm.lid:0 shm.cnt:1
5		**n_2.lid:0 n_2.cnt:1**	$CAS_c(1,0,1)$✗		shm.lid:0 shm.cnt:1
6		ContactLeader(n_1.lid:0)✗	**n_3.lid:0 n_3.cnt:1**	$CAS_c(1,0,1)$✗	shm.lid:0 shm.cnt:1
7	$CAS_l(0,0,1)$ ✓	$CAS_c(1,1,2)$✓	ContactLeader(n_3.lid:0)✗	**n_4.lid:1 n_4.cnt:1**	**shm.lid:1 shm.cnt:2**
8	BecomeLeader()	$CAS_l(1,0,2)$✗	$CAS_c(2,1,2)$✗	ContactLeader(n_4.lid:0)✗	shm.lid:1 shm.cnt:2
9		**n_2.lid:1 n_2.cnt:2**	**n_3.lid:1 n_3.cnt:2**	$CAS_c(2,1,2)$✗	shm.lid:1 shm.cnt:2
10				**n_4.lid:1 n_4.cnt:2**	shm.lid:1 shm.cnt:2
	$\text{Total}_{\text{read}}$:2 $\text{Total}_{\text{write}}$:2	$\text{Total}_{\text{read}}$:7 $\text{Total}_{\text{write}}$:1	$\text{Total}_{\text{read}}$:6 $\text{Total}_{\text{write}}$:0	$\text{Total}_{\text{read}}$:6 $\text{Total}_{\text{write}}$:0	

Table 5.10. Example with four nodes

$$\mathrm{T}_{\mathrm{LE,win}} = 2 * T_{read} + 2 * T_{CAS}$$

In the second case, the node fails performing the first CAS instruction, that means, comparing its counter stored in its local memory to the counter stored in the shared memory. So the node reads three times from the shared memory. So the time complexity used for one node when it fails in the first CAS of the algorithm is:

$$\mathrm{T}_{\mathrm{LE,loose}} = 3 * T_{read}$$

In the third case, the node fails when comparing its leader id stored in its local memory to the leader id stored in the shared memory. Thus the node wins the first CAS instruction and fails performing the second CAS instruction comparing the leader id. Thus the node reads and updates one time due to the first CAS instruction (counter) and reads three times due to fail comparing the leader id using the second CAS instruction. So the time complexity used for one node when it fails in the second CAS of the algorithm is:

$$\mathrm{T}_{\mathrm{LE,loose2}} = 4 * \mathrm{T}_{\mathrm{read}} + \mathrm{T}_{\mathrm{CAS}}$$

Table 5.10 shows an example of four nodes, and how many reads and writes they need to elect a leader.

If the system has n nodes, then the leader election algorithm has one node that becomes leader; one or the zero node that change the counter in the shared memory and does not become leader; n-2 nodes updates their local data. Lemma 7.1.6 shows that the nodes in the worst case perform the algorithm twice as a maximum. The node that changes the counter without becoming leader is the node that performs the algorithm twice. While executing the first algorithm, the node fails when comparing the counter stored in its local memory and the counter stored in the shared memory. Therefore, the node updates its local memory before another node becomes leader, which means that the node only updates its local counter and the leader id in its local memory is the leader id of a crashed leader. So the node performs leader election again. In this second execution, the node increases the counter in the shared memory, but fails when comparing the leader ids stored in its local memory to the leader id in the shared memory.

As a result, the System-Wide Time Complexity (SWTC) for the leader election algorithm is:

$$T_{\mathrm{SWTC}} = 1 * T_{\mathrm{LE,win}} + 2 * (n-2)T_{\mathrm{LE,loose}}) + 1 * T_{\mathrm{LE,loose2}}$$

where n are the nodes that compete to become leader and:

$$T_{\text{LE,win}} = 2 * T_{\text{read}} + 2 * T_{\text{CAS}};$$

$$T_{\text{LE,loose}} = 3 * T_{\text{read}};$$

$$T_{\text{LE,loose2}} = 4 * T_{\text{read}} + 1 * T_{\text{CAS}}$$

Consequently, the system-wide time complexity in the worst case is:

$$T_{\text{SWTC}} = [2 * T_{\text{read}} + 2 * T_{\text{CAS}}] + [(2 * (n - 2)) * 3 * T_{\text{read}}] + [4 * T_{\text{read}} + T_{\text{CAS}}]$$

As a result:

$$T_{\text{SWTC}} = [6 * (n - 1)] * T_{\text{read}} + 3 * T_{\text{CAS}}$$

In conclusion, the system-wide time complexity in the worst case has a constant time related to the CAS instruction, but the time related to read information from the shared memory is lineal depending the number of nodes of the system.

5.7 Chapter summary

The use of the leader election mechanism is beneficial for distributed systems with a centralised node that has a specific work, given that the addition of this fault tolerance mechanism resolves the problem if the leader has crashed. The leader election chooses a new leader using only non-busy nodes to elect a new leader, discriminating busy nodes, therefore our approach does not force all nodes of the system to participate in the leader election. Consequently, a busy node is not interrupted to participate in the leader election, permitting that the busy node finishes its work without interruptions. After the busy node has finished its work, the node updates the previous data related to the old leader with the new leader data stored in the shared memory. As a result, the leader election mechanism proposed avoids nodes time waiting until the other system nodes are finished and other possible interruptions, having an advantage in terms of time against other algorithms (presented in Section 3), favouring a faster method and better performance.

At the end of this chapter, it is presented the evaluation of the leader election proposed in terms of space and time complexity per node, as well as system-wide time complexity. Given that the leader election algorithm proposed works asynchronously, the system-wide time complexity for our algorithm, in the worst case, might depend on the number of nodes that are part of the system. Regarding to the space and time complexity per node, our algorithm is constant independently of the number of nodes used in the system.

Chapter 6

Log-Based Rollback-Recovery

6.1 Model

In LPEL, a node works as a **leader** of the system and the remaining nodes are **workers**. The leader is responsible for sending tasks to the workers. Then the workers perform tasks and return the task to the leader when the task is finished.

This thesis focuses on avoiding transient faults. Transient faults can be recovered if the determinants are saved in stable storage [70]. For this reason, a stable storage is created for each node that works as an independent local buffer to other nodes in the system. Therefore, each worker has its own stable storage, called a **Log Buffer** (LB).

It is assumed that the data stored in the stable storage are failure-free, which means that we are sure that the information stored until that time is correct. We need this assumption to be sure that the information stored in th stable storage are related to the information that it is happening in the system.

Each stable storage is represented by a buffer and each line of the stable storage, independently of stable storage being for a worker or a leader, has a boolean flag to indicate if the register stored in that line is obsolete or not, and a register related with a non-deterministic event. This boolean inside each line of stable storage can only be changed by the garbage collection mechanism.

Furthermore, a new role is created for the leader, called **Fault Tolerance Control** (FTC), to help the restore mechanism used for our rollback-recovery mechanism proposed. This role is responsible for managing the relevant information of the leader and supports the workers that are performing the restore mechanism. At the beginning, we thought that a separate node acquires this role, but we decided that the leader can do it and it was not necessary to use a separate node. Therefore, the leader changes its role to FTC when a message is received from a worker that performs the restore method.

It is important to remember that fault detection is not a topic of this thesis, but there a few papers related to that topic [68, 11]. We focus on proposing two mechanisms (leader election and log-based rollback-recovery) to resolve the fault after it appears in the system.

In our log-based rollback-recovery algorithm, when a worker performs the restore mechanism, the communication between the leader and this worker stops until the worker finishes the restoring method.

The restore mechanism consists in restarting the task performed by the worker and uses the registers stored in its LB to restore the last state of the worker before the fault appeared. If the necessary register is not in the LB, the worker sends a message to the FTC (Leader) to ask if any worker has the demanded register.

Last but not least, we propose a garbage collection mechanism to remove the data stored in FTC and the LB that is not require anymore. This mechanism is automatic and not periodical, which means that it is activated when one register is stored in stable storage (FTC or LB). Then the mechanism checks all the registers stored in the stable storage and removes the obsolete registers. The Garbage Collection Mechanism (Section 6.1.4) defines the precondition and the rules followed for each register to be removed.

Figure 6.1 shows an example of how stable storages are distributed among two workers and one leader.

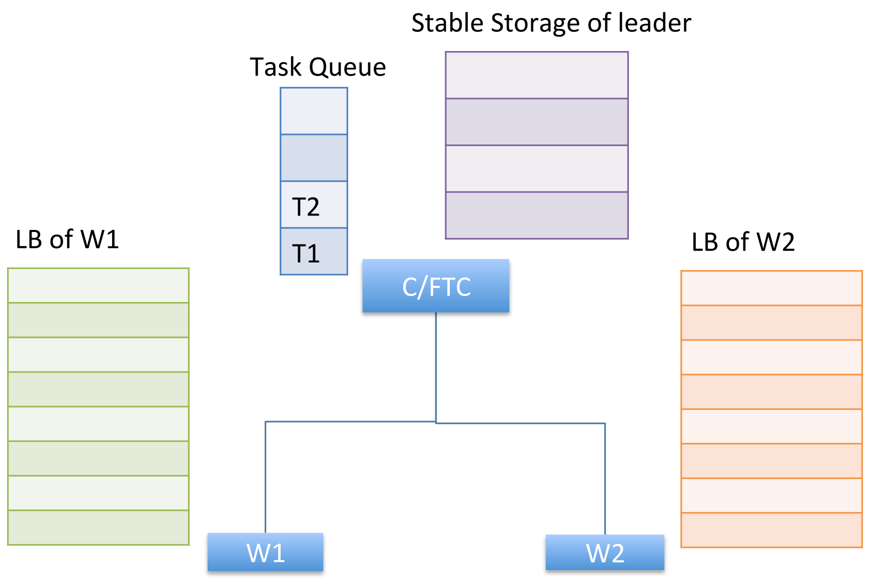


Figure 6.1. Example of log-based rollback-recovery with two workers

This section is composed of 5 subsections: Section 6.1.1 describes how each worker stores the registers in the LB. Section 6.1.2 explains the *Fault Tolerance Control* role. The Section 6.1.3 discusses the property of LPEL to block tasks and how our algorithm overcomes this issue. The Section 6.1.4 shows how the *Garbage Collection* mechanism removes any useless registers in the stable storages. Finally, Section 6.1.5 presents the restore mechanism used by our algorithm.

6.1.1 Log Buffer

As described in Section 3, during failure-free operations, each node stores the determinants of all the non-deterministic events observed in its stable storage [84].

The LB is normally only read and written by the associated worker. The FTC can read and write the LB only when the worker has failed and the worker has not yet recovered.

A non-deterministic event is created when one of the five following events occurs:

1. A worker receives a task.
2. A task inside a worker consumes information from the input stream.
3. A task inside a worker writes information to the output stream.
4. A task is finished.
5. A worker returns a task to the leader.

We propose storing in the LB of the worker the following registers related to the previously described non-deterministic events:

1. <**START**, Task>
2. <**READ**, TaskId, StreamId, InputData, Counter>
3. <**WRITE**, TaskId, StreamId, OutputData, Counter>
4. <**BLOCKED**, TaskId, IsTaskReceived>
5. <**END**, TaskId, IsTaskReceived>

where the comma (,) is used to separate the information. *START*, *READ*, *WRITE*, *BLOCKED* and *END* are tokens used by the registers to represent the non-deterministic event that affects the worker.

<**START**, Task>

> This register is created after the worker has received a task from the leader. Every time a worker receives a task, the worker creates this register in its LB.
>
> This register is composed of a token *START* and a copy of the task that contains the checkpoint of its state before the worker performs it. Since that we have developed a log-based rollback-recovery mechanism proposed for LPEL and S-Net, the copy of the task is stored using the *lpel_taks_t* structure in LPEL.

lpel_taks_t is a data structure used by LPEL to store the task that will be performed by the workers. If an error is detected while the worker is performing the task, then the restore mechanism uses the checkpoint stored in this register to restart the task.

<**READ**, TaskId, StreamId, InputData, Counter>

A task, performed by a worker, reads input data from the input stream by creating a copy in the local buffer of the worker and removing the data from the stream. This event is stored using the register with the token *READ*.

The second parameter of this register is the identifier of the task. The restore mechanism can use this identifier to relate this register with the task with the same identifier. The third parameter is the identifier of the stream where the data are read. The forth parameter is a copy of the data read. Last but not least, the fifth parameter is the *Counter* parameter and is a counter. This counter is used to differentiate among the readings of the same data.

For example, a task wants to read data from a stream, and after reading these data, they are removed from the stream. There other data with the same information exist in the stream. Therefore, the task can re-read the same data from the stream. So this parameter is used to differentiate between the data read this second time and the first time, and it is used for the restore mechanism if a failure appears. This parameter is explained in detail in the FTC section.

We store this register before the task reads the data from the stream because the data are removed from the stream after the task makes a copy in the local buffer of the worker. Let's imagine that the register is created after reading the data. If a failure appears between this reading and the creation of the register, the data that the task tries to read is lost and cannot be used by the restore mechanism. Therefore, this register stores a copy of the data without removing them from the stream, and before the reading action is performed.

If an error appears before the task reads the data and after the worker stores a copy of the data in the register, FTC is responsible for overcoming any incongruences. Further information is found in the FTC section.

<**WRITE**, TaskId, StreamId, OutputData, Counter>

Before the task writes data to the output stream, the worker stores this register in the LB. Let's imagine that the register is created after writing the data. If a failure appears between the writing and creating the register, the data remain in the stream, but there is no proof about the writing for the restore mechanism. When the task is performed for the second time by the restore mechanism, these data are written for the second time in the stream. This problem can be avoided if the register is stored before writing the data and using FTC to know if the data have been written or not.

The token of this register is *WRITE* because it is related with the non-deterministic event when the task inside the worker writes data to the output stream.

The *Counter* parameter is similar to the counter parameter in the *READ* register and counts how many times the task writes the output data. With this parameter, the task can write each data per stream more than once.

This register is similar to the *READ* register, and instead of storing the input data, it stores the output data that the task writes to the output stream.

<**BLOCKED**, TaskId, IsTaskReceived>

LPEL allows a task to be blocked, in which case the worker stops executing of the task and returns the task to the leader. We created this register to allow this feature in our proposal which informs the restore mechanism that the task has been blocked and maybe is in another worker. Furhter information about this feature is found in the Block Task section. The *BLOCKED* register is created before the task is returned to the leader.

The *IsTaskReceived* parameter is a Boolean parameter. When this register is created, this parameter is marked as *FALSE* and it changes when the worker receives the confirmation message from FTC that the leader has received the task.

<**END**, TaskId, IsTaskReceived>

This register is created after the worker finishes the task and before the worker sends the task to the leader. This register is the last task-related register and the following registers have no a reference to the task.

This register is composed of the *END* token, the identifier of the task and the *IsTaskReceived* parameter. The token and the identifier are used by the garbage collection mechanism to remove any useless registers related to this task.

The *IsTaskReceived* parameter is a Boolean parameter that is *FALSE* when the *END* register is created. The worker changes this parameter to *TRUE* after the worker receives the confirmation message from FTC that the leader has received the task. This third parameter is used to avoid failures in the middle of garbage collection. The Restore Mechanism section contains further information about this issue.

Algorithm 8 Structure of the registers related to the log buffer

```
typedef struct log_worker_start
{
    lpel_task_t task;
} log_worker_start;

typedef struct log_worker_read
{
    int task_id;
    int stream_id;
    void * input_data;
    int counter;
} log_worker_read;

typedef struct log_worker_write
{
    int task_id;
    int stream_id;
    void * output_data;
    int counter;
} log_worker_write;

typedef struct log_worker_blocked
{
    int task_id;
    int is_task_received;
} log_worker_blocked;

typedef struct log_worker_end
{
    int task_id;
    int is_task_received;
} log_worker_end;
```

The structures of these registers are shown in Algorithm 8. There are five different types of structures, one per register stored. The *READ* and *WRITE* register have a *void* pointer to the point the copy of the input data and the output data, respectively.

6.1.2 Fault Tolerance Control

As said in the beginning of this chapter, a new role in the system is created called Fault Tolerance Control (FTC) that helps the worker to restore its state when the restore mechanism is called. This new role helps the system to be more robust.

In our design a single node serves as both FTC and leader. It is necessary to differentiate between the leader role and the FTC role. The leader role is responsible for distributing the available tasks to available workers.

The leader plays the FTC role when the leader receives a message from a worker that performs the restore mechanism. Additionally, FTC is responsible for returning the task to the leader if one worker suddenly crashes and the task in which the worker is involved cannot be returned to the leader. For example, if a worker crashes in the middle of executing a task and it is not possible to recover the worker using our proposed log-based rollback-recovery, then FTC gets the task stored in the LB of the crashed worker and inserts the task back into the leader's task queue, to assign it later again to another worker.

FTC uses the registers stored in the leader's stable storage to help recover a failed worker when this worker, after having become functional again, calls the restore mechanism. This is explained in detail in the Section 6.1.5.

A non-deterministic event that affects the leader appears when:

1. The leader sends a task to a worker.

2. The leader receives a task from the worker that has finished performing its assigned task.

3. The stream is written by a worker.

In addition, FTC is affected by a non-deterministic event related to the writing of the stream. Therefore, we have three different non-deterministic events that affect the leader and FTC.

The list below shows the registers created by the FTC to store the previously explained non-deterministic event:

1. <**SENT**, WorkerId, Task>

2. <**RECEIVED**, WorkerId, TaskId, IsFinished>

3. <**STREAM**, TaskId, StreamId, Data, Counter>

where the comma (,) is used to separate information. *SENT*, *RECEIVED* and *STREAM* are tokens created to know if the register is created when the leader sends a task or receives a task, or when a worker writes data to a stream. *WorkerId* is the identifier used by the leader to contact the worker. *TaskId* is the identifier of the task received and *Task* is a copy of the task sent to a worker. *StreamId* is the identifier of the stream where *Data* are stored. *Counter* is a counter that counts how many times data have been written to a stream. *IsFinished* is a Boolean parameter that allows the system to know if the received task is finished in the worker or if the task is blocked and needs to be finished.

<**SENT**, WorkerId, Task>

> The leader creates and stores this register before the leader sends a task to a worker with the identifier *WorkerId*. The *Task* parameter is a backup of the task. FTC uses this backup if the worker *WorkerId* crashes and the leader cannot recover the task from the worker.

<**RECEIVED**, WorkerId, TaskId, IsFinished>

> The leader creates and stores this register after the leader receives the task *TaskId* from the worker *WorkerId*. *WorkerId* and *TaskId* are identifiers that help log-based rollback-recovery to find the relevant registers related to the worker and the task, respectively.
>
> *IsFinished* is a Boolean parameter that the garbage collection mechanism uses to know if the worker has completely performed the task (*TRUE*) or if the task is blocked when the worker performs this task (*FALSE*).
>
> When a *RECEIVED* register is stored with *IsFinished* as *TRUE*, FTC searches all the *RECEIVED* registers with the same *TaskId* and sends a message to all the workers in those registers which inform that another worker is finishing the task *TaskId*. Further information about the blocked task can be found in the Block Task Section.

<**STREAM**, TaskId, StreamId, Data, Counter>

> FTC stores this register to the leader's stable storage after FTC receives a copy of the *Data* from the worker that writes *Data* to the stream *StreamId*. Therefore, this register contains the *TaskId* of the task that writes the data *Data* in stream *StreamId*.
>
> The *Counter* parameter is a counter used to know how many times these data have been written by a task. This parameter allows the tasks to write more than once each lot of data per stream.

We need to choose carefully when the register is stored, before or after the action has been completed, to avoid inconsistency with the data stored into the LB. For example, if the leader fails after storing a *SEND* register, but before sending the task, the information in stable storage is incorrect because the task has not been sent. When a leader fails, it is called the leader election method. After electing a new leader, the FTC processes the inconsistencies caused by the previous leader: e.g. the previously described example. The structures of these registers are shown in Algorithm 9.

Algorithm 9 Structures of the registers stored in the leader's stable storage

```
typedef struct log_leader_sent
{
    int worker_id;
    lpel_task_t task;
} log_leader_sent;

typedef struct log_leader_received
{
    int is_finished;
    int worker_id;
    int task_id;
} log_leader_received;
typedef struct log_leader_stream
{
    int task_id;
    int stream_id;
    void * data;
    int counter;
} log_leader_stream;
```

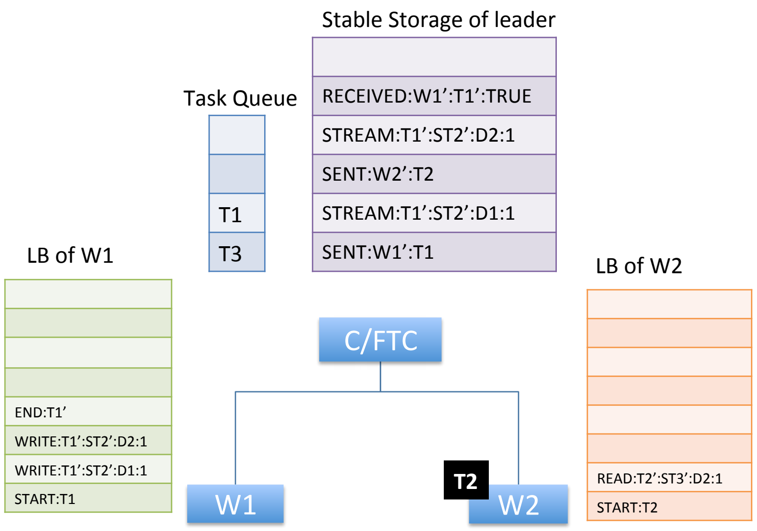


Figure 6.2. Example of the leader storing the register in its stable storage.

Figure 6.2 is an example of a system with two workers and one leader. ID' represents the identifier of a worker (e.g. W1') or a task (e.g. T1') or a stream (e.g. ST2'), whilst ID represents an exact data copy of a task (e.g. T1) or data (e.g. D1) read from or written to the stream. In this example, the leader sends task $T2$ to worker $W2$. Previously the leader sends and receives task $T1$ to worker $W1$. The garbage collection mechanism is deactivated in this example. The data stored in the stream cannot be read by a task until FTC confirms that the data in the stream are correct.

Additionally, FTC is responsible for comparing the data stored in streams with the data from the workers stored in the LB. When a task writes data in a stream, FTC receives a notification from the stream that new data are written. In parallel, the worker where the task is performed sends a message with a copy of the data written to the FTC, and the FTC creates a *STREAM* register with the data received from the worker. Next, the FTC compares the data received from the worker and the data stored in the stream. If they are equal, the FTC does nothing. if they are different, the FTC changes the data in the stream with the data received from the worker.

If the worker crashes and the FTC does not receive the message with the data from the worker, the FTC reads the information stored in the LB of the worker to check the data in the stream.

The task restarted by the restore mechanism tries to read or write to a stream, which leads the worker to ask the FTC whether the *READ* or *WRITE* register, that the task tries to perform, finished before the failure appeared or if the information stored in the LB is incorrect because the failure appears before the reading or writing is done. The FTC displays a different behaviour if it receives a *READ* or *WRITE* register from the worker.

If a task is performed by the restore mechanism, wants to read data from the stream and the LB of the worker has a *READ* register to that data, the worker sends a message to the FTC to check if the data were removed from the stream. If the data are still in the stream, the FTC removes them from the stream, otherwise the FTC does nothing. Irrespectively of removing the data from the stream or not, the FTC sends a confirmation message to the worker.

When the restore mechanism performs a task and this task wants to write data in a stream, the worker sends a message to FTC to know if the data are written or if the register stored in the LB of the worker is correct. This avoids a failure to appear after a *WRITE* register is stored, but before the data are written to the stream.

6.1.3 Block Task Event

In LPEL, tasks can be blocked at any time by creating a non-deterministic event called **task-blocked** [100]. This blocking is caused by the LPEL application because the tasks in LPEL have different priorities. So the tasks with a high priority are performed urgently, which means, LPEL blocks a task with less priority that a worker performs and changes the task with less priority into that worker's task with a higher priority.

We have to keep in mind that LPEL cannot block a task while it is reading from the input stream or writing to the output stream.

The following list shows the possibilities of a task in the system when the task is blocked:

- The task starts in worker W_1 and moves through one worker or more until the task is finished by another worker other than worker W_1.
- The task starts in worker W_1 and moves through one worker or more, but the task is finished by worker W_1.

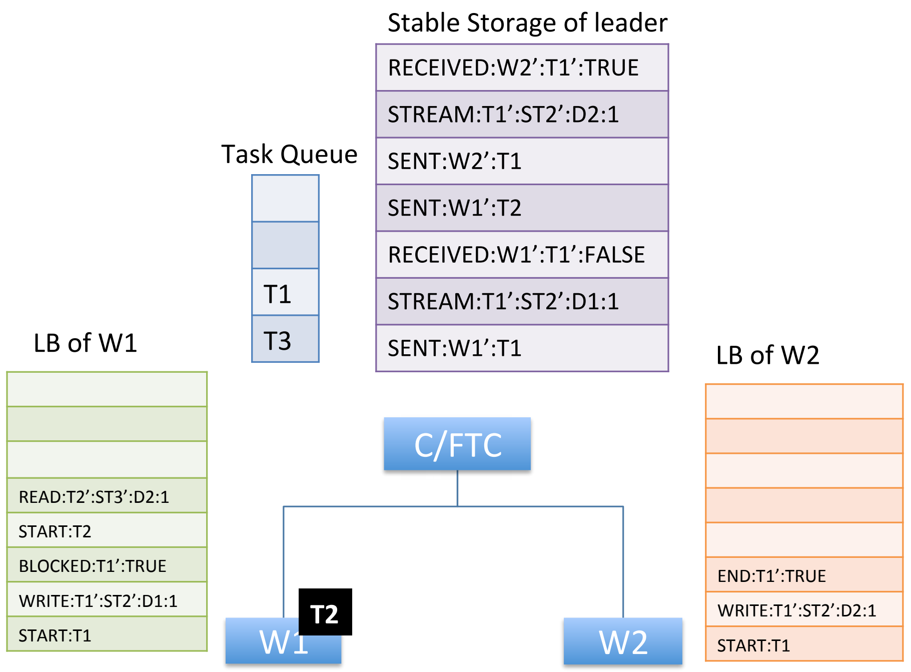


Figure 6.3. Example of a task blocked and finished in another worker

The task starts in one worker and finishes in another

Figure 6.3 shows a system composed of two workers (W_1 and W_2), one leader and three tasks (T_1, T_2 and T_3). We focus on task T_1. At the beginning, T_1 starts its execution in worker W_1. T_1 writes the data D_1 to output stream ST_2 and then the system blocks T_1 and the leader sends T_2 to W_1 because T_2 is more critical than T_1. Therefore, T_1 is returned to the leader and the *RECEIVED* register of T_1 in leader's stable storage is marked as not finished (*FALSE*). After the leader receives T_1 from the W_1, the FTC sends a confirmation message to W_1 confirming that the leader has received T_1. Then the leader sends T_2 to W_1. W_1 receives (*START*) and performs T_2. The leader sends T_1 to worker W_2 and W_2 creates the *START* register in the LB. T_1 continues its execution from the last point before it is blocked. T_1 writes data D_2 to output stream ST_2 and finishes its execution while it is performed by W_2.

In this example, the leader saves in its stable storage the information about the worker where the task was executed with the *IsFinished* parameter as *FALSE*. When the task is returned as finished, the leader sends a message to all the workers where the task was executed. So W_1 receives a message that T_1 was finished by worker W_2. The garbage collection mechanism of W_1 uses this message to remove any useless registers. In this example, the garbage collection mechanism is deactivated.

The leader's stable storage stores a *RECEIVED* register with the *IsFinished* parameter as *FALSE* every time a task is blocked. For this reason, the changes made by a task are always recorded in the stable storage.

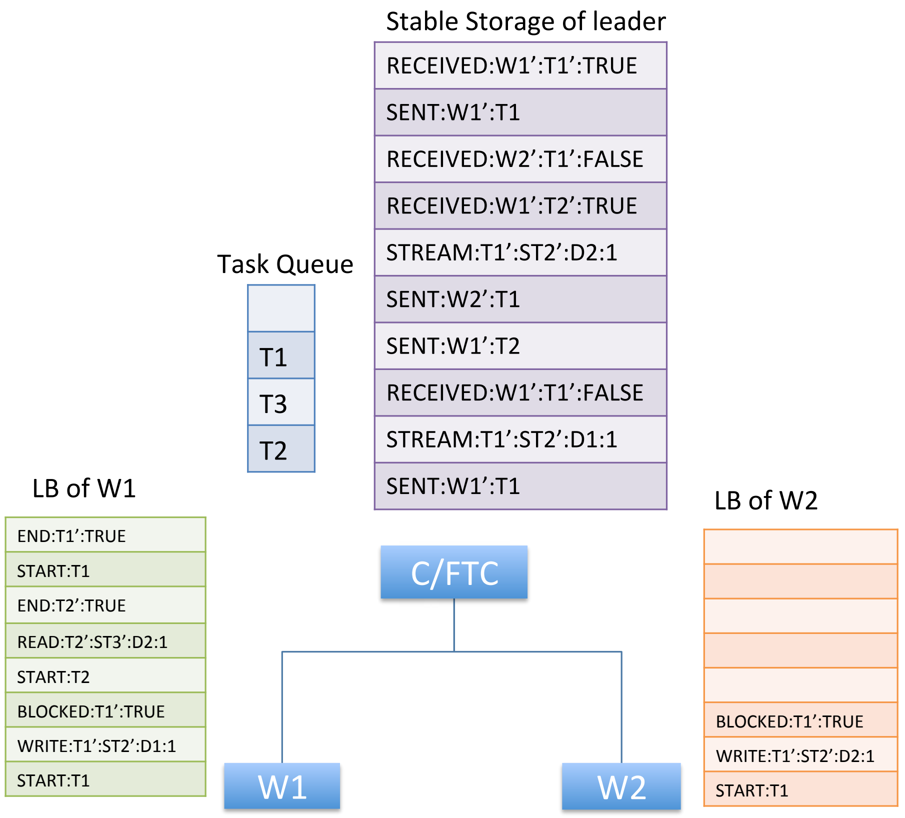


Figure 6.4. Example of a task blocked and finished in the first worker where it was executed

Task is started by one worker and is performed by other workers, but is finished by the first worker

In Figure 6.4, task T_1 starts in worker W_1. T_1 is blocked after writing data D_1 to output stream ST_2 and goes to worker W_2. Then T_1 writes the data D_2 to output stream ST_2 and is blocked again. This time T_1 returns to W_1. At this time, W_1 stores a *START* register with the last checkpoint of the task before the task is reperformed by W_1. Finally, T_1 is finished by W_1 and returned to the leader.

The parameter *IsFinished* related to T_1 is marked two times as FALSE into the stable storage of the leader, being T_1 blocked by W_1 and W_2. The second time T_1 is performed by W_1, the execution of T_1 is finished and *IsFinished* is marked as TRUE. Then this example of a task blocked can be solved in the same way as in the previous case using the parameter *IsFinished* to manage where the task was executed.

6.1.4 Garbage Collection Mechanism

The stable storage used in this presented approach is implemented as double-linked lists. As a result, we need to create a garbage collection mechanism to remove the nodes that contain obsolete information related to the log-based rollback-recovery algorithm.

As you can see in the bibliography [54, 128] we can find different kinds of garbage collection mechanisms, nevertheless, these mechanisms have a big execution time overhead. For this reason, we created a specific garbage collection to minimize the execution time overhead presented in the other mechanisms. Therefore, the garbage collection mechanism implemented has a low computation overhead and does not modify the output of the application [75].

At the beginning, the double-linked list has only two nodes: *HEAD* and *TAIL*. The *HEAD* node is the first node of the double-linked list and the *TAIL* node is always the last node of this list. When a new node is created, it is added between the previous node of the *TAIL* node and the *TAIL* node. The created node contains a register used for the log-based rollback-recovery algorithm.

The garbage collection mechanism is automatic and not periodical. This mechanism is activated when a node with a new register is added in stable storage (FTC or LB) or when the *IsTaskReceived* parameter of the *END* register changes to *TRUE*.

The garbage collection mechanism checks all the nodes stored on the double-linked list, except the node most recently stored. Then the register stored in each node has a precondition that allows garbage collection to remove it. If the precondition is accomplished, the node is removed.

The idea is to add a node between the last node and the *TAIL* node. Garbage collection checks nodes by starting with the previous node of the node most recently added. After checking a node, the garbage collection checks the previous nodes of the checked node. Then the mechanism continues checking nodes until the checked node becomes the *HEAD* node.

The following list shows the garbage collection precondition of each register:

- **START**: When the *END* register is stored in the LB independently of the *IsTaskReceived* parameter or when the worker receives a message from FTC that the task was finished by another worker.

- **READ**: Same as the *START* register.

- **WRITE**: This register has two preconditions that must be accomplished. The first precondition is the same as the *START* register. The second precondition is accomplished when the worker receives a confirmation message from FTC that the data were stored without problems in the stream. As long as one of these two preconditions is not accomplished, the *WRITE* register cannot be removed.

- **BLOCKED**: This register has two preconditions. The first precondition is the same as the *START* register. The second precondition is this register is the last register removed from the stable storage. That means, the *START*, *READ* or *WRITE* registers related to the task must not be in the stable storage to accomplish this precondition. When both preconditions are accomplished, this register is removed.

- **END**: This register has two preconditions. The first precondition is related to the confirmation message sent by FTC. Once the leader has received the task from the worker, the FTC sends a confirmation message to this worker. The other precondition is related to the registers of the task. This means that the *END* register needs to wait until all the other registers of the task are removed from the worker's LB. Then the *END* register is the last register related to the task removed from the LB.

- **SENT**: When the *RECEIVED* register related to the task is stored with the *IsFinished* parameter as *TRUE.*

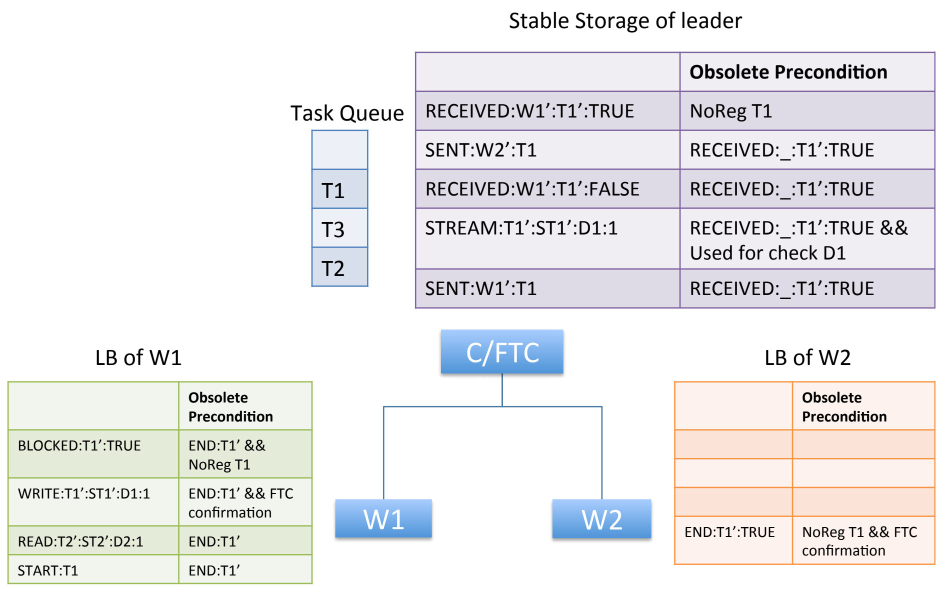
- **RECEIVED**: This register presented two cases. On the one hand, the register can be stored with *IsFinished* as *FALSE* and, on the other hand, the *IsFinished* parameter can be stored as *TRUE.* A *RECEIVED* register with *IsFinished* as *FALSE* will be removed when another *RECEIVED* register related to the same task is stored with *IsFinished* parameter as *TRUE.*

  A *RECEIVED* register with *IsFinished* parameter as *TRUE* will be the last register removed related to the task in the leader's stable storage. Accordingly, this register is removed when there are not other registers related to the task.

- **STREAM**: This register has two preconditions. The first precondition is the same as the *SENT* register. The second precondition is accomplished after the FTC compares the data in this register to the related data in the stream.

The *RECEIVED* and *BLOCKED* registers receive special attention in the garbage collection mechanism. The *RECEIVED* register with *IsFinished* as *FALSE* and the *BLOCKED* register with *IsTaskReceived* as *TRUE* or *FALSE* are the only registers that do not activate the garbage collection mechanism.

Yet if the *RECEIVED* register has the *IsFinished* parameter as *TRUE*, garbage collection is performed.



Stable Storage of leader

	Obsolete Precondition
RECEIVED:W1':T1':TRUE	NoReg T1
SENT:W2':T1	RECEIVED:_:T1':TRUE
RECEIVED:W1':T1':FALSE	RECEIVED:_:T1':TRUE
STREAM:T1':ST1':D1:1	RECEIVED:_:T1':TRUE && Used for check D1
SENT:W1':T1	RECEIVED:_:T1':TRUE

LB of W1

	Obsolete Precondition
BLOCKED:T1':TRUE	END:T1' && NoReg T1
WRITE:T1':ST1':D1:1	END:T1' && FTC confirmation
READ:T2':ST2':D2:1	END:T1'
START:T1	END:T1'

LB of W2

	Obsolete Precondition
END:T1':TRUE	NoReg T1 && FTC confirmation

Figure 6.5. Example of the garbage collection mechanism

Figure 6.5 shows an example of the necessary precondition of each register. In this example workers W_1 and W_2 perform task T_1. The figure includes the *obsolete precondition* columns that contain the condition necessary for registers to be removed. These columns are not represented in the following figures of this thesis. In the figure, the *START* register in the *LB of W1* is removed when W_1 receives the message from FTC that T_1 finished in another worker(T_2). *NoReg T1* means that the condition is accomplished when all the registers related to T_1 in the LB have been removed. A confirmation message from FTC is represented as the *FTC confirmation* in the figure. *Used for check D1* is accomplished when FTC compares the data stored in the stream to the data of the *STREAM* register.

6.1.5 Restore Mechanism

In this section, we explain the restore mechanism implemented to restore the state of the system using the checkpoint and log registers stored. The proposed restore mechanism is based on restarting and replaying [22] the task that the worker performs according to the information stored into the worker's LB.

The worker that performs the mechanism stops the task execution and rollback to last checkpoint the task in the worker. So the execution performed between the last checkpoint stored and the previous time before the fault was detected is lost. After restarting the task, the worker uses the information stored into its LB to help the task recover the state before the fault has appeared.

When a worker runs the restore mechanism, the remaining system workers are not aware of the fault of the worker that runs the restore mechanism. The remaining workers are not disturbed by the fault of a worker thanks to the independent failure property of distributed systems [129].

Our restore mechanism begins by reading the last register stored in the LB. Figure 6.6 shows the behaviour of the restore mechanism depending on the first read register. In case the first register read by the restore mechanism is an *END* register, then the restore mechanism checks the *IsTaskReceived* parameter of the register. If the *IsTaskReceived* parameter is *FALSE*, then the worker sends the task to the leader. Therefore the restore mechanism finishes and the worker waits for the confirmation message from the FTC. It can occur that the leader receives the same task twice. In this situation, FTC resends a confirmation message to the worker and the task received this second time is removed since that it was already received.

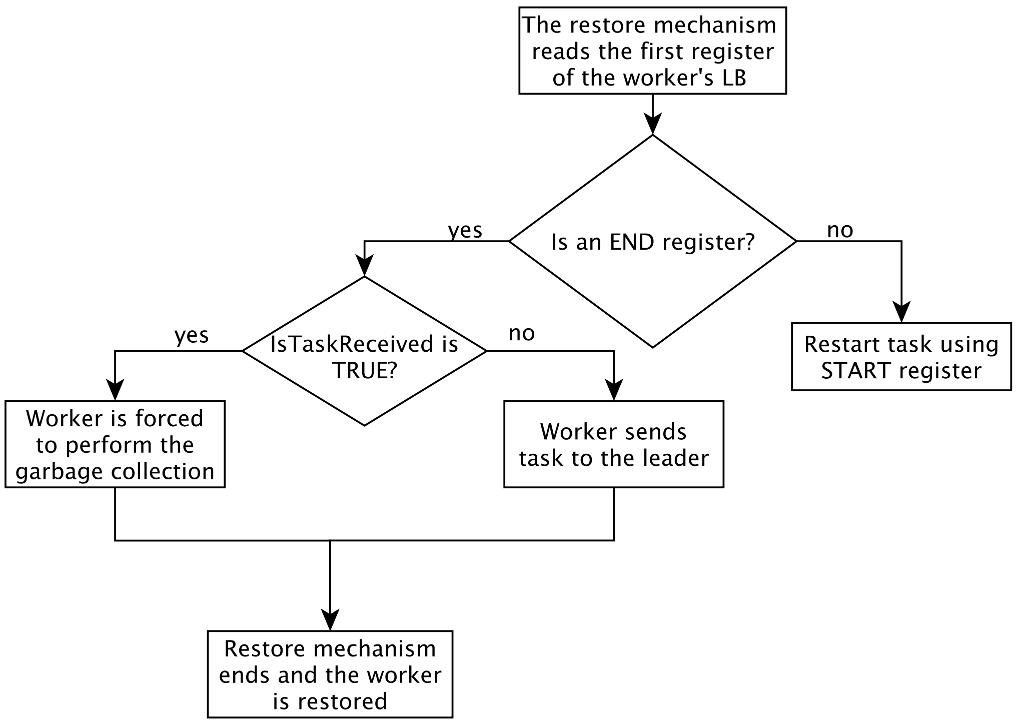

The restore mechanism reads the first register of the worker's LB
Is an END register?
yes
no
IsTaskReceived is TRUE?
yes
no
Restart task using START register
Worker is forced to perform the garbage collection
Worker sends task to the leader
Restore mechanism ends and the worker is restored

Figure 6.6. Restore mechanism behaviour after reading the first register in the workers' LB

When the *IsTaskReceived* parameter is *TRUE*, the restore mechanism checks if there are any other registers related to the same *END* register task. If affirmative, the restore mechanism forces the worker to perform garbage collection. After finishing the garbage collection, the restore mechanism finishes and the worker is restored.

On the other hand, if the first register read by the restore mechanism is a *BLOCKED* register, the restore mechanism checks the *IsTaskReceived* parameter of the register. On the condition that this parameter is *TRUE*, the restore mechanism finishes. However, when the parameter is *FALSE*, the restore mechanism forces the worker to send the task to the leader. After this, the restore mechanism finishes and the worker awaits the confirmation message from the FTC. The performance of the restore mechanism for *BLOCKED* register is similar to the *END* register, nevertheless, the garbage collection is not called for the *BLOCKED* register.

In case the first register read by the restore mechanism is not an *END* or *BLOCKED* register, then the task that the worker performs is stopped and the task is restarted using the last checkpoint stored in the *START* register of the task. The restore mechanism chooses the last *START* register stored into the LB, obtaining the last stored checkpoint of the task.

The restore mechanism reads the checkpoint stored in the *START* register and restarts the task with this checkpoint. Next the task is reperformed by the restore mechanism supervision. During this second execution, the restore mechanism uses the *READ* and *WRITE* registers to recover the task. The *READ* and *WRITE* registers stored in the LB are used to avoid that task reads or writes data to the input or output stream. The registers stored in the LB are not modified and are removed by the restore mechanism.

During this second task execution, the worker performs the task by starting from the checkpoint stored in the *START* register. Yet when the task needs to read from the input stream, the task searches a *READ* register with the data in the LB instead of the input stream. If the needed data are in the LB, the task reads the data from the register and the worker sends a message to FTC which indicates that the data were read by the task. This message contains the task identifier and is used by FTC to search the data in the stream. If the data is in the stream, FTC removes the data to the stream, because the *READ* register contains the copy of this data and the task has already read this data but not removed to the stream..

If the data that the task wants to read are not in the LB, the worker sends a message to the FTC to ask if someone has the required data. Then FTC uses the information stored into the leader's stable storage to ask other worker that had been performing previously the same task.

When the task needs to write a data to an output stream, it checks if there is a *WRITE* register in the LB related with the data that the task wants to write. If the data are already in a register stored in the LB, the worker sends a message to the FTC to ask if the data were written before the failure affected the worker. Then the FTC checks if there is a *STREAM* register related to the data asked for by the worker. If there is a *STREAM* register, the FTC replies to the worker that the data were written before the failure appeared. So the task does not write the data to the output stream. If there is no a *STREAM* register, the FTC replies to the worker that the data were never written in the stream. As a result, the task writes the data to the output stream and the worker creates the *WRITE* register with the recently written data.

If there is no *WRITE* register in the worker's LB related to the data that the task wants to write, the worker sends a message to the FTC to ask if another worker wrote these data in the stream.

Until this point, *READ* and *WRITE* await a message from the FTC. The FTC uses the information stored in the leader's stable storage to send a message to all the workers that worked with the task. When the message from FTC reaches a worker that previously worked in the task, the worker checks to see if there is a register stored in its LB related to the data requested by the FTC. If the worker's LB has a register with the demanded data, the worker sends a message with a copy of the stored data. If the worker does not have a related data register, the worker sends a message to the FTC saying that it does not have the requested data.

The FTC receives all the messages from the workers that previously worked with the task. If there is a message with the demanded data, the FTC sends this message to the affected worker, otherwise, FTC sends a message to the affected worker that the demanded data were never stored in another LB. It should be pointed out that FTC immediately sends the message that contains a copy of the data stored in another LB to the affected worker, and it does not await messages from all the workers.

In both cases, *READ* and *WRITE*, if no register related with the demanded data exits in another LB, the register is created, and the task reads or writes to the stream. Figure 6.7 shows how the task reads or writes the data when the restore mechanism performs the task.

The FTC is also responsible for avoiding incorrect data appearing in the leader's stable storage. These incorrect data can appear in the following situations:

- The leader fails after storing a *SENT* register and before sending the task.
- The leader fails after receiving the task and before storing a *RECEIVED* register.

When a leader is elected with leader election, the FTC checks the leader's stable storage. If there is a *SENT* register and there is no *RECEIVED* register with the *IsFinished* parameter as *FALSE* related to the same task, then FTC asks the worker of the *SENT* register for the task status. As a result, the worker can reply with the following messages:

- NOT_RECEIVED: in this case, the FTC sends the task to the worker related to the *SENT* register because the leader had failed before immediately after the register was created.
- WORKING: FTC does nothing because the worker performs the task.

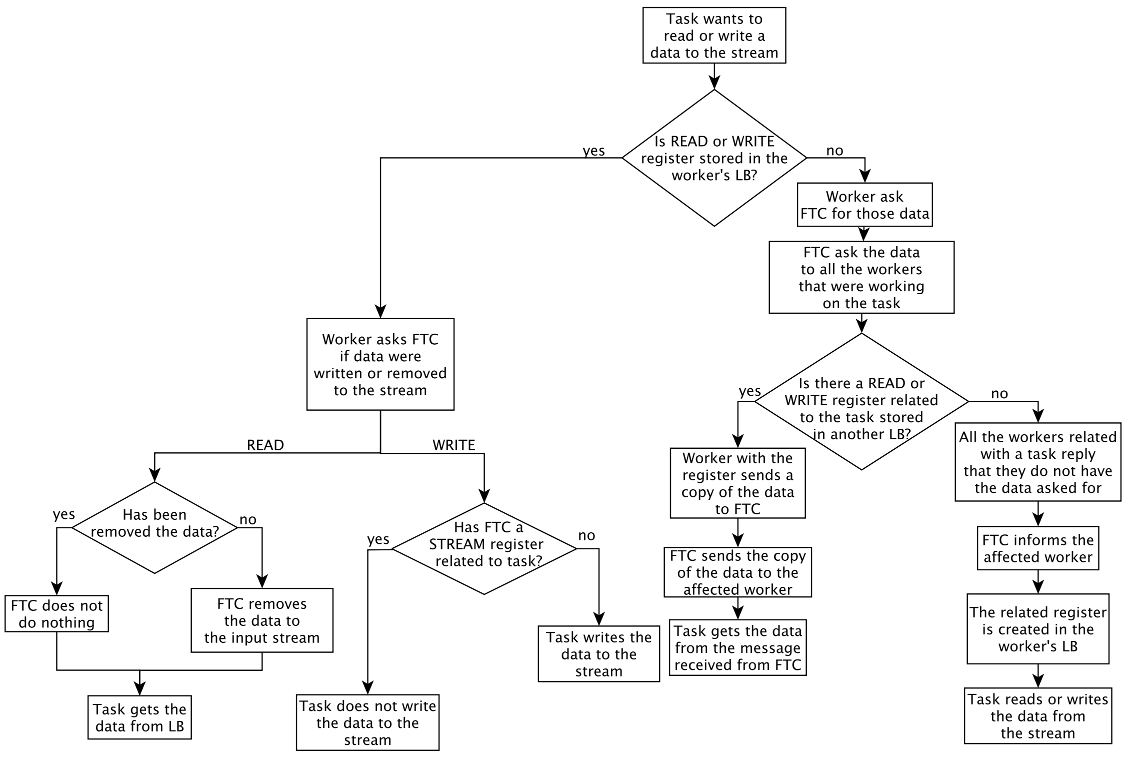

Task wants to read or write a data to the stream
Is READ or WRITE register stored in the worker's LB?
yes
no
Worker ask FTC for those data
FTC ask the data to all the workers that were working on the task
Worker asks FTC if data were written or removed to the stream
Is there a READ or WRITE register related to the task stored in another LB?
yes
no
READ
WRITE
Worker with the register sends a copy of the data to FTC
All the workers related with a task reply that they do not have the data asked for
Has been removed the data?
yes
no
Has FTC a STREAM register related to task?
yes
no
FTC sends the copy of the data to the affected worker
FTC informs the affected worker
FTC does not do nothing
FTC removes the data to the input stream
Task writes the data to the stream
Task gets the data from the message received from FTC
The related register is created in the worker's LB
Task gets the data from LB
Task does not write the data to the stream
Task reads or writes the data from the stream

Figure 6.7. Behaviour of the task that tries to read or write the stream when it is performed by the restore method

- FINISHED: FTC checks if the task is in the leader's task queue. If the task is not in the task queue, FTC copies the task from the *SENT* register to the task queue. If the task is in the task queue, a *RECEIVED* register related to the task and the worker is created in stable storage, and garbage collection is performed with this new register.

6.2 Examples of Log-Based Rollback-Recovery

A practical example of log-based rollback-recovery for a distributed system with streaming communication is explained in this section.

This practical example has one leader (C/FTC), two workers (W_1 and W_2) and three tasks (T_1, T_2 and T_3).

In the example, ST_1 is the output stream of T_1 and the input stream of T_2, ST_2 is the output stream of T_2 and T_1, and the input stream of T_3 and ST_3 is the output stream of T_3.

Regarding the task, T_1 twice writes D_1 to stream ST_1 and D_2 to stream ST_2. T_2 reads D_1 from ST_1 and writes D_3 to ST_2. T_3 reads D_2 and D_3 from ST_2 and writes D_4 to ST_3. T_2 is not sent to a worker until D_1 is in ST_1. T_3 is not sent to a worker until D_2 and D_3 are in ST_2.

Figure 6.8 shows the state of the system before tasks are performed.

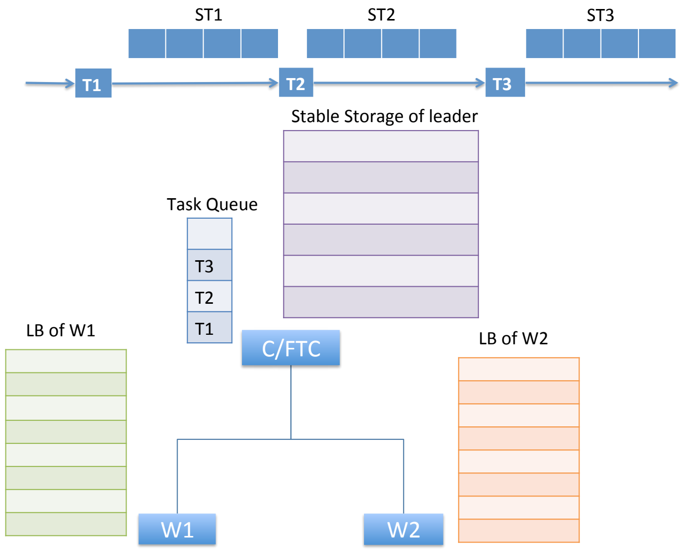


Figure 6.8. Initial state of the log-based rollback-recovery example

At the beginning, the leader sends T_1 to W_1 to be performed, but before sending it, FTC creates a $SENT$ register related to it in the leader's stable storage. W_1 receives T_1, creates a $START$ register of T_1 and performs T_1. While T_1 is performed by W_1, T_1 writes D_1 in ST_1 twice and W_1 sends two messages about D_1 to FTC. FTC stores two $STREAM$ registers related to D_1 and checks both D_1. Both D_1 stored in the stream are saved without errors. After D_1 writes the second time in ST_1, a fault is detected in W_1. So W_1 starts the restore mechanism.

When D_1 is written in ST_1, the leader sends T_2 to W_2. W_2 receives T_2 and stores a $START$ register of T_2. W_2 performs T_2 while W_1 performs T_1. T_2 reads D_1 from ST_2, but before W_2 stores a $READ$ register related to D_1.

Figure 6.9 shows the exact time when the fault is detected in W_1. D_1 is represented as strike-through because it was removed from ST_1 when T_2 reads it. The lines removed by garbage collection or removed because the data were read by a task are represented as strike-through lines in the figure, and they do not appear in the following figures.

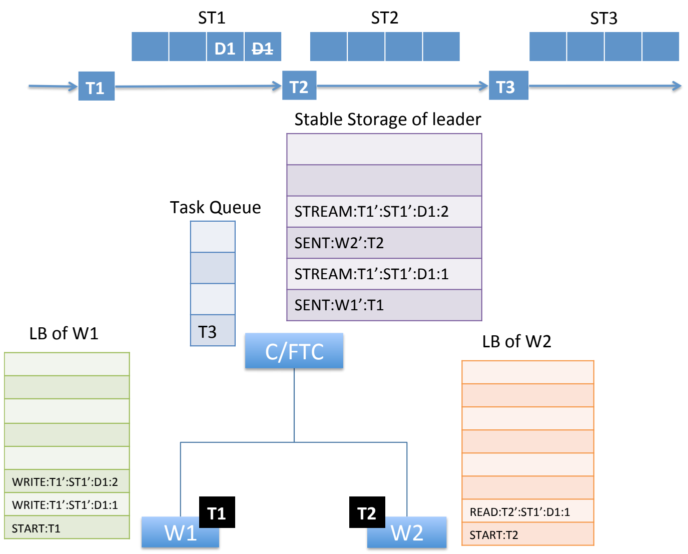


Figure 6.9. Snapshot of the log-based rollback-recovery example when a fault is detected in W_1 and the restore mechanism is called

The restore mechanism searches the $START$ register stored in the LB of W_1 and restarts T_1 using the checkpoint stored in the $START$ register. W_1 performs T_1 from the checkpoint stored by the $START$ register. Meanwhile, W_2 finishes performing T_2, because T_2 is not returned to the leader, the garbage collection mechanism is not called yet. Figure 6.10 shows a snapshot of this time. T_1 is represented in red because the task is performed a second time by the restoring mechanism.

The snapshot shown in Figure 6.11 represents the exact time when the leader has received T_2 and FTC has sent the confirmation message to W_2. Then the garbage collection removes the useless register from the LB of W_2 and the leader's stable storage.

T_1 wants to write D_1 in ST_1 for the first time, but it is avoided by the restore mechanism because a register about D_1 exists in ST_1 with counter 1.

Furthermore, the restore mechanism forces W_1 to ask FTC whether another worker that performed T_1 wrote D_1 in ST_1 with counter 1. W_1 asks FTC because W_1 does not know if the error appeared after the register was stored and before the data were written.

If the register is in the LB of W_1, but is not in the leader's stable storage, T_1 writes D_1 to ST_1 with counter 1, without modifying the register related in the LB, while W_1 sends a copy of the data to FTC.

In this case, FTC replies that the data were written in ST_1, so T_1 does not write D_1 to ST_1.

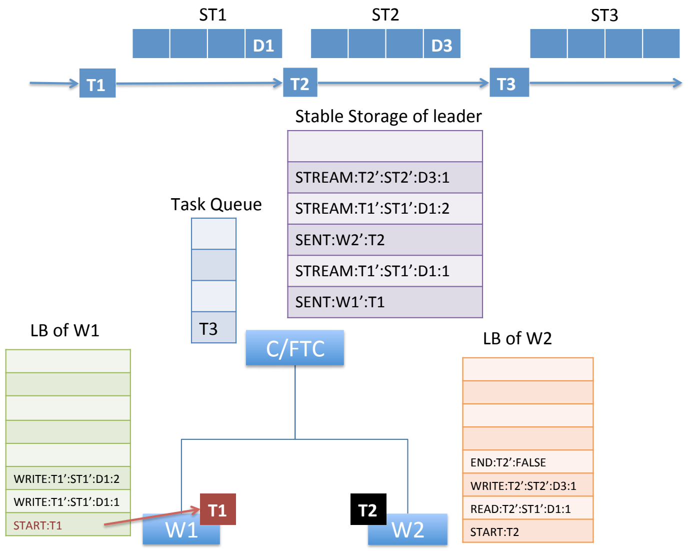


Figure 6.10. Snapshot of the log-based rollback-recovery example when T_1 is performed by the restore mechanism

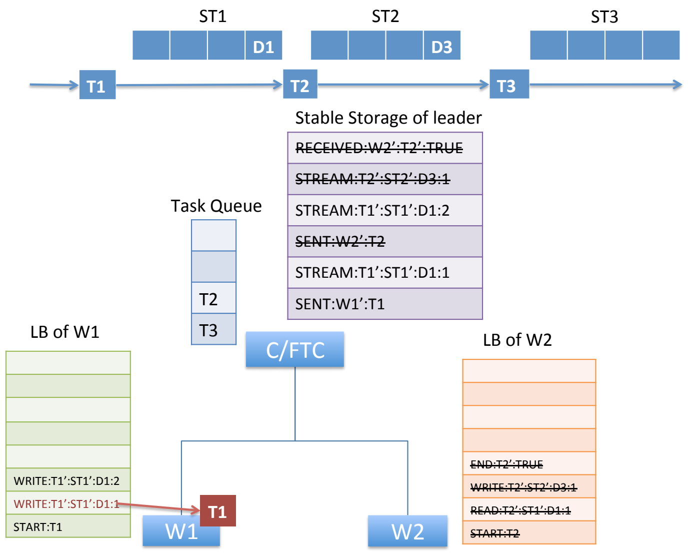


Figure 6.11. Snapshot of the log-based rollback-recovery example when T_2 finishes while T_1 is re-performed a second time

Figure 6.12 shows how T_1 tries to write D_1 a second time in ST_1, but the restore mechanism avoids writing in ST_1 because D_1 was written before the fault was detected. W_1 asks FTC to confirm if it knows that D_1 was written in ST_1. T_3 is not yet performed because D_2 is not in ST_2.

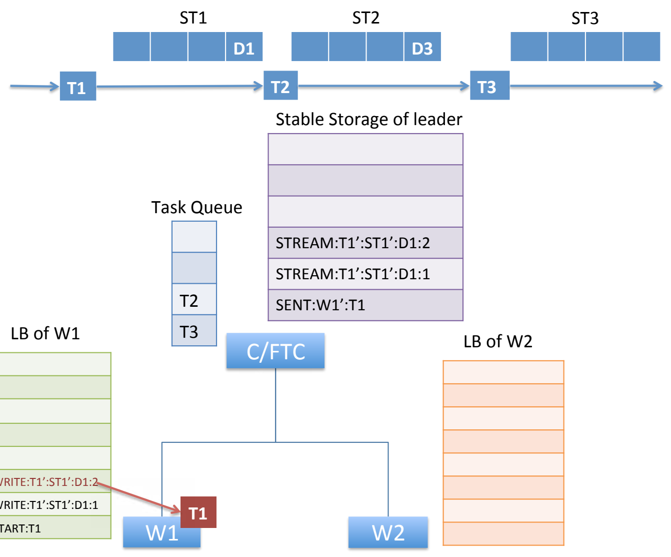


Figure 6.12. Snapshot of the log-based rollback-recovery example when T_1 tries to write D_1 for the second time

T_1 wants to write D_2 in ST_2, but the LB of W_1 does not have a register related to these data. As a result, W_1 asks FTC if another worker wrote D_2 with T_1.

In this case, FTC replies to W_1 that no writing was performed by another worker. Therefore, W_1 creates the register in its LB and sends it, and T_1 writes D_2 to ST_2. Afterwards, T_1 writes D_2, W_1 sends a copy to FTC, and FTC creates a *STREAM* register with the received data. It is possible to perform T_3 because D_2 and D_3 are in ST_2.

Then the leader sends T_3 to W_2 and W_2 creates the *START* register related to T_3, as it is shown in Figure 6.13.

Figure 6.14 shows that this time T_1 is completed without faults and T_3 continues its execution. In the above figure, T_1 is still to be sent to the leader, but garbage collection can remove any useless registers in the LB of W_1.

T_1 returns to the leader, FTC creates the *RECEIVED* register with the *IsFinished* parameter as *TRUE* and sends the confirmation message to W_1. The garbage collection is activated by the *RECEIVED* register removing the register related to T_1 into the leader's stable storage. Note that the last removed register about T_1 from the leader's stable storage was the *RECEIVED* register with the *IsFinished* parameter as *TRUE*.

Figure 6.15 represents the snapshot when $W1$ awaits its next task and garbage collection finishes with the leader's stable storage.

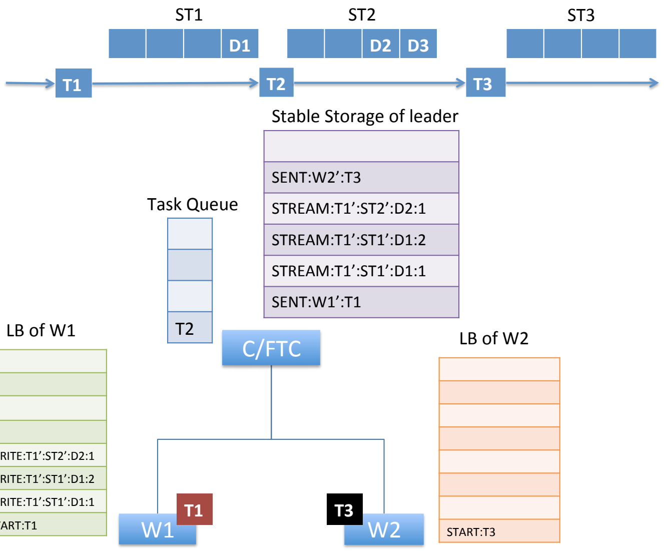


Figure 6.13. Snapshot of the log-based rollback-recovery example when the leader sends T_3 to W_2

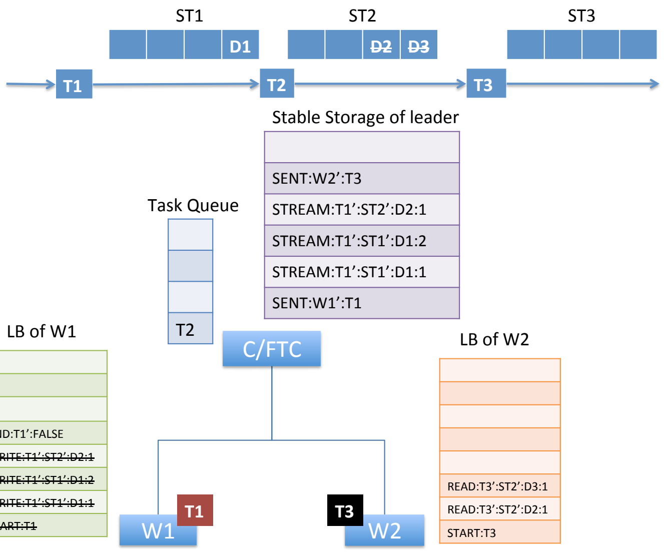


Figure 6.14. Snapshot of the log-based rollback-recovery example when W_1 finishes $T1$

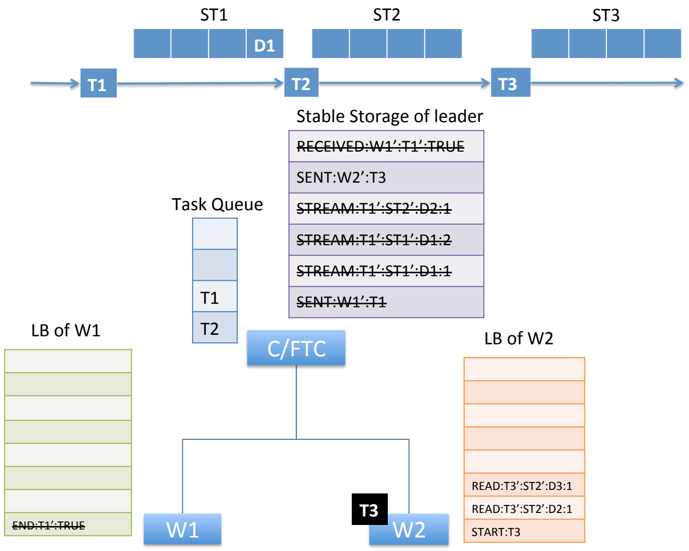


Figure 6.15. Snapshot of the log-based rollback-recovery example when the leader receives T_1

At the end, T_3 writes D_4 to ST_3 and T_3 is returned to the leader. Figure 6.16 shows the state of the system after performing the example.

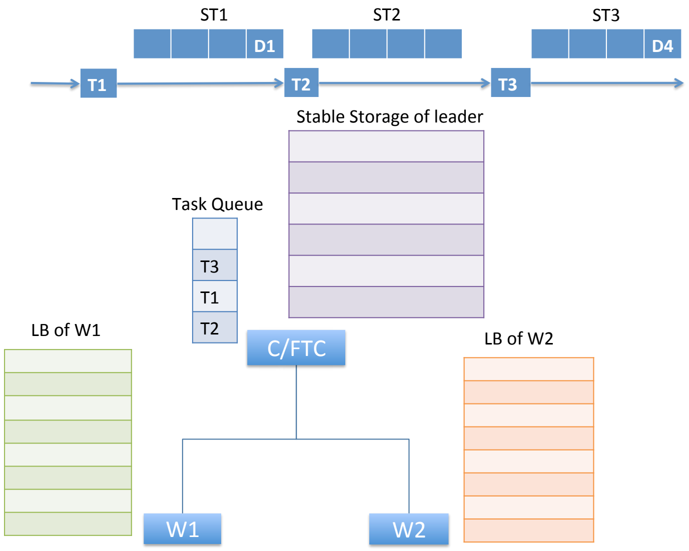


Figure 6.16. State of the system after performing the log-based rollback-recovery example

With this example, we demonstrate that the garbage collection is working and the stable storage are never full with obsolete information. Additionally, the example shows how the log-based rollback-recovery proposed can restore a worker after it is affected by a failure without interrupting other workers.

6.3 Chapter Summary

The log-based rollback-recovery mechanisms can be very beneficial for distributed systems to restore the system after a failure has appeared. For this reason in this chapter, the log-based rollback-recovery proposed is focus on restoring distributed systems with streaming communication affected by transient faults.

We also created a new role in the system called FTC. This new role helps an affected worker to restore its state before it was affected by a failure, using the registers stored into the leader's stable storage. In addition, the FTC checks if the data stored into the streams is correct, and supervises the tasks sent and received by the leader to handle the possible errors that can affect the workers.

An advantage of our mechanism proposed is that it only restores the affected worker, without interrupting or affecting other workers.

Another benefit is that in our algorithm the LPEL characteristic of blocking task event has been added. So, the smooth running of our log-based rollback-recovery proposed is not disturbed by this feature.

Last but not least, we developed a garbage collection mechanism for our log-based rollback-recovery mechanism that allows the use of a low overhead garbage collection mechanism, in comparison with other existing garbage collection mechanism presented in Section 3.

Chapter 7

Correctness of the Proposed Fault Tolerance

This chapter shows the definitions, theorems and proofs of the leader election presented in Section 5 and the log-based rollback-recovery proposed in Section 6 in this thesis.

7.1 Correctness of Leader Election

In the following section, we provide some formal definitions and proofs of the correctness of the leader election method.

We use a global well-clock time in our definitions and proofs, This clock is defined by us, it is not related to another system clock and it is used to define which events happen after others. If there are two events and one of them occurs before the other one, we can know with our global clock, that the time (t_1) of the event happens before and in consequence, it is smaller than the other time (t_2) event.

$$\text{time } t_1, t_2 \implies t_1 < t_2$$

7.1.1 Definitions

We provide a list of basic definitions, used to formally describe the correctness of the created leader election algorithm:

- $\mathit{isLeader}(\mathbf{n}, \mathbf{t})$ or $\mathit{IL}(\mathbf{n}, \mathbf{t})$: node n is the leader of the system at time t, then the shared memory (shm) stores the id of node n at time t.

$$\mathit{isLeader}(n, t) \iff (\mathit{time} = t \implies n.\mathit{id} = \mathit{shm.lid})$$

- $\mathit{isCounter}(\mathbf{c}, \mathbf{t})$ or $\mathit{IC}(\mathbf{c}, \mathbf{t})$: counter c is the counter stored in the shared memory at time t.

$$\mathit{isCounter}(c, t) \iff (\mathit{time} = t \implies c = \mathit{shm.cnt})$$

- $\mathit{hasLeader}(\mathbf{n}, \mathbf{n_0}, \mathbf{t})$ or $\mathit{HL}(\mathbf{n}, \mathbf{n_0}, \mathbf{t})$: node n has leader n_0 at time t in its local memory.

$$hasLeader(n, n_0, t) \iff (time = t \implies n.lid = n_0.id)$$

- *hasCounter*$(\mathbf{n}, \mathbf{n_C}, \mathbf{t})$ or $HC(\mathbf{n}, \mathbf{n_C}, \mathbf{t})$: at time t, node n has counter n_C in its local memory.

$$hasCounter(n, n_C, t) \iff (time = t \implies n.cnt = n_C)$$

- *believesLeader*$(\mathbf{n}, \mathbf{t})$ or $BL(\mathbf{n}, \mathbf{t})$: at time t, node n believes it is the leader, denoted by having its local leader id $n.lid$ indicating itself:

$$believesLeader(n, t) \iff (time = t \implies n.lid = n.id)$$

7.1.2 Correctness

In this section, correctness arguments for our leader election algorithm are provided. The assumptions for the proof of the correctness of this work are:

Assumption 7.1.1 *Each node has a unique identifier and knows its own identifiers, but it does not need to know the identifier of the other nodes.*

Assumption 7.1.2 *There is no priority among the nodes that take part in leader election. Any available node can become leader after the leader crashes.*

Assumption 7.1.3 *The communication timeout is longer than the maximal time a that complete leader election method takes.*

Assumption 7.1.4 *Failed node may reintegrate in the future.*

Lemma 7.1.5 *(Negative Result) Even given the assumption that a given time exists with a maximum of one node which believes it is leader, Algorithm 6 does not ensure for all future times that there will be at maximum one node that believes it is leader. Formally, the following correctness property is violated:*

$$(\exists t.\ \nexists n_1, n_2.\ (n_1 \neq n_2) \land BL(n_1, t) \ \land\ BL(n_2, t)) \implies$$
$$(\forall t_1.\ \nexists n_3, n_4.\ (t_1 \geq t) \land (n_3 \neq n_4) \land BL(n_3, t_1) \ \land\ BL(n_4, t_1))$$

Proof 7.1.5

Proof that Lemma 7.1.5 is sufficient to show a scenario in which the correctness property is violated. Our scenario arises from the case of the nodes that perform different leader election rounds at the same time, with failed nodes allowed to reintegrate (Assumption 7.1.4).

We assume that at time t, two nodes (n_1 and n_2) are prepared to perform Algorithm 6, and node n_L is the current failed leader. We assume that node n_1 was leader before n_L became leader, and that n_1 reintegrated after it failed as leader to act since that time as a worker with n_L as its leader. Since n_L failed, node n_1 looks for a successor of n_L as a leader. We further assume that node n_2

was busy for a longer period, and that it missed the transition of the leader role from n_1 to n_L. Since n_1 is now a normal worker, it does not response to n_2's leader service request. Thus we have the following formal states at time t that results from multiple overlapping leader election rounds:

$$\exists t.\ \exists n_1, n_2, n_L.\ IL(n_L, t) \land HL(n_1, n_L, t) \land HL(n_2, n_1, t) \land (n_1 \neq n_L) \quad \text{(eq.7.1)}$$

We assume that at after rounds, time t_1 ($t_1 \geq t$) node n_1 becomes leader because n_1 performs $CAS(shm.lid, nde.lid, nde.id)$ in line 3 of Algorithm 6 before node n_2 does:

$$\exists t_1.\ [eq.7.1] \land\ (t_1 > t) \land BL(n_1, t_1) \land IL(n_1, t_1) \land HL(n_2, n_1, t_1) \quad \text{(eq.7.2)}$$

Later at time t_2 ($t_2 \geq t_1$), node n_2 performs the CAS in line 3 of Algorithm 6 and becomes leader, while node n_1 is still also alive and acts as leader. The result is an inconsistent state where the system has a new leader (n_2) and, at the same time, another node (n_1) also believes to be the leader:

$$\exists t_2.\ [eq.7.2] \land (t_2 > t_1) \land BL(n_1, t_2) \land IL(n_2, t_2) \land BL(n_2, t_2) \quad \text{(eq.7.3)}$$

In conclusion, this proof shows that Algorithm 6 cannot guarantee that at the same time at maximum one node believes to be the leader. Note that the shown scenario is based on the ability to reintegrate failed nodes once they recover from failure (Assumptions 7.1.4) □

Lemma 7.1.5 shows that the inconsistent state for Algorithm 6 is caused by the ABA problem. The ABA problem (Section 5.4) appears when a node reads the shared memory twice: shared memory has the same value for both reads, and the value is used to indicate that nothing has changed. However, the other nodes executed between the two reads may change the variable, perform different work and then change the variable to its previous value. Then the first node believes that the shared memory value has not changed.

Resolving the ABA problem avoids that the system can have two leaders at the same time, what could cause a split in the system resources.

Lemma 7.1.6 *If there is a time t with at maximum one node believes that it is leader, then Algorithm 7 preserves that property for all future times:*

$$uniqueLeader(t) \iff (time = t) \land \nexists n_1, n_2.\ (n_1 \neq n_2) \land BL(n_1, t) \land BL(n_2, t) \quad \text{(eq.7.4)}$$

$$\forall t_1 . \exists t.\ (uniqueLeader(t) \land (t_1 \geq t)) \implies uniqueLeader(t_1) \quad \text{(eq.7.5)}$$

Proof 7.1.6 This proof is based on individually showing the behaviour of each node that performs the leader election of Algorithm 7. We assume a node n performs the first CAS to test whether the current leader count matches it local view, i.e., $n.cnt = shm.cnt$. This $CAS(shm.cnt, n.cnt, n.cnt + 1)$ can return as either *true* or *false*, which we analyse separately:

a) $CAS(shm.cnt, n.cnt, n.cnt + 1)$ **returns *false*:** in this case, before the first CAS we had $n.cnt \neq shm.cnt$, which means that at least one other node already started the Algorithm 7 before and passed the first CAS instruction. Node n, in this case, merely concludes that another node has become leader and updates its local view of *n.cnt* and *n.lid*. However, there are still two cases to distinguish:

a.1) Another competing node has already completed the second CAS instruction of Algorithm 7:
In this case, node n has the node with id *shm.lid* as its new leader.
$\boxed{\text{Result: Node } n \text{ finds a new leader in this round}}$

a.2) No competing node has already completed the second CAS instruction:
In this case, *shm.lid* still indicates the failed leader. Node n will, therefore, later recognise that the assumed leader is not available and will restart leader election. In the second round, node n will encounter either case a.1 or b.1 rather than case a.2, based on the assumption that one of the competing nodes will then also have completed the second CAS instruction. Therefore, in the second round, node n cannot perform case b.2, because the leader id stored in its local memory is different than the leader id stored in the shared memory.
$\boxed{\text{Result: Node } n \text{ will find a new leader in the next round (via case a.1 or b.1)}}$

b) $CAS(shm.cnt, n.cnt, n.cnt + 1)$ **returns *true*:** in this case, node n will run the second CAS to test whether the leader id in the shared memory is still the id of the failed leader, i.e., $n.lid = shm.lid$. Once again, there are two cases to consider:

b.1) $CAS(shm.lid, n.lid, n.id)$ **returns *false*,** which means another node has already become new leader and node n will only update its local view of *n.cnt* and *n.lid*, and the leader election for node n will finish with node n remaining a worker.
$\boxed{\text{Result: Node } n \text{ finds new leader in this round}}$

b.2) $CAS(shm.lid, n.lid, n.id)$ **returns** *true*, which means that node n has become the new leader, and the leader election for node n finishes with node n assuming the leader role.
$\boxed{\text{Result: Node } n \text{ becomes leader in this round}}$

Concluding from these four subcases, it follows that at maximum one node n can become leader via case b.2 with the second CAS instruction in the same

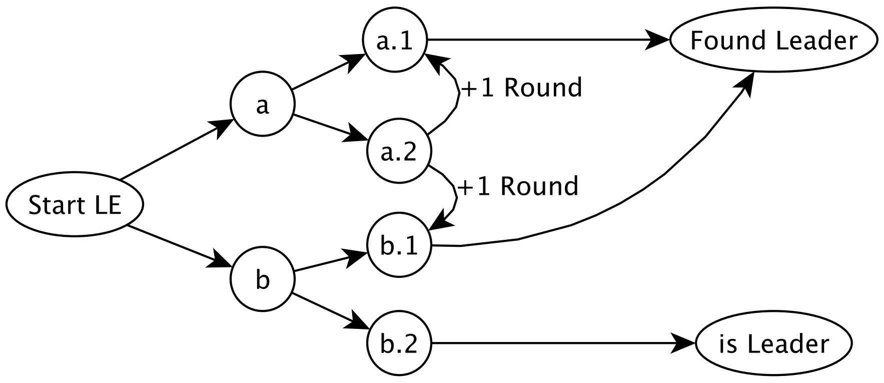


Figure 7.1. Reaction of nodes to CAS instructions in the leader election

leader election round, where the current leader election round is defined as via *shm.cnt*. All the nodes n in the same leader election round have the same *n.lid* value before leader election. However since the first node succeeded via case b.2, it follows that all the other nodes of the same round can fail via case a.1 or a.2. If the node is in case a.2, the node performs again the leader election and, in this second round, the node can run the leader election via case a.1 or b.1. Any other nodes *n'* that participate at the same time with a previous leader election round (i.e., *n'.cnt* < *shm.cnt*) will automatically be ruled out as leader via case a) and will eventually accept the new leader (those in case a.2, but only through the next round). All the cases demonstrated in this proof are represented in Figure 7.1. □

As it is shown in the proof of Lemma 7.1.6, the nodes that achieve case a.2 need to run twice the leader election. For this reason we can define the Corollary 7.1.7.

Corollary 7.1.7 *A node has to run Algorithm 7 at most twice to finish leader election.*

Lemma 7.1.8 *The counter stored in the local memory of the nodes (n_c) equals or is less than the counter in the shared memory:*

$$\forall t.\ \forall n.\ \exists n_c, c.\ HC(n, n_c, t) \wedge IC(c, t) \wedge (n_c \leq c)$$

Proof 7.1.8 At the beginning, the counter in the nodes and the counter in the shared memory equal zero (*n.cnt* = 0 and *shm.cnt* = 0). Leader counters are modified only by the leader election (Algorithm 7). The leader count is used for $CAS(shm.cnt, n.cnt, n.cnt + 1)$ in line 3 of Algorithm 7. CAS can either return $true$ or $false$, with the following results:

a) $CAS(shm.cnt, n.cnt, n.cnt + 1)$ **returns *true*:**

$\boxed{\text{Result: The counter in the shared memory is increased by one.}}$

b) $CAS(shm.cnt, n.cnt, n.cnt + 1)$ **returns *false*:**

$\boxed{\text{Result: The counter in the shared memory is not modified.}}$

Therefore, the local counter of a node is never increased in Algorithm 7, but changes when the node updates its local information from the shared memory. Consequently the local counter of the node never has a greater value than the counter of the shared memory. □

Lemma 7.1.9 *The space complexity in the shared memory used on each node is constant.*

Proof 7.1.9 The shared memory and each node store two variables:

1. An integer with the leader id
2. An integer of how many leader elections were performed (counter)

In our proposal, the number of nodes does not affect the number of stored variables. Then the algorithm needs the same two variables independently if the system has four or four hundred nodes. Therefore, the space complexity is constant. □

We will also proof that Algorithm 7 is wait-free. According with the definition of wait-free, an algorithm can be defined as wait-free when every node performing the algorithm has a limited number of steps to complete the algorithm. [65, 97].

Lemma 7.1.10 *The leader election algorithm is wait-free.*

Proof 7.1.10 In our leader election algorithm, there is
There is one loop in the communication algorithm, but no loop in the leader election algorithm. Corollary 7.1.7 shows how the node has to run the algorithm a maximum of two runs in order to finish leader election. Consequently, the number of steps for a node to become a leader or to update its local information is finite. So, our leader election proposed has a bound of number of operations and, as a result, we can say that the leader election mechanism presented is wait-free. □

Additionally, we will proof that Algorithm 7 is lock-free. The Lock-Free property guarantees that at least one node is progress on its work [66]. In theory this means that a method may take an infinite amount of operations to complete, but in practice it takes a short amount, otherwise it won't be much useful. Therefore, an algorithm is Lock-Free if it guarantees that infinitely often some node calling this method finishes in a finite number of steps. Then all wait-free algorithms are lock-free.

Lemma 7.1.11 *The leader election algorithms is lock-free.*

Proof 7.1.11 In Lemma 7.1.10, it is proof that the leader election algorithm is wait-free, as a result, the algorithm presented is lock-free. □

7.2 Correctness of Log-Based Rollback-Recovery

In the following section, we provide some formal definitions and proofs of the correctness of the log-based rollback-recovery algorithm proposed.

7.2.1 Definition

In the following section, we provide a list of basic definitions used to formally describe the correctness of the created log-based rollback-recovery algorithm, where e denotes a non-deterministic event:

- *Depend*(**e**): The set of nodes (workers and/or the leader) need the non-deterministic event e to continue performing their tasks.

  In the example presented in Section 6.2, T_3 depends on D_2 and D_3 to start. Then the worker that performs T_3 is affected by the non-deterministic events that create D_2 and D_3.

- *Log*(**e**): the set of workers and/or the leader that stored a register related to event e in their stable storage.

- *Stable*(**e**): The non-deterministic event e is converted to a register and the register is stored in the node's stable storage.

7.2.2 Correctness

This section contains the correctness proof of the log-based rollback-recovery algorithm presented in this thesis. The following assumptions are used as proof of correctness:

Assumption 7.2.1 *Log-based rollback-recovery is a method used to recover from a fault, and not to detect it. Therefore, additional mechanism have to be provided for detecting a fault and, consequently performs log-based rollback-recovery to solve a fault appearing.*

Assumption 7.2.2 *The communication between the leader and workers is error-free, i.e., there are no communication errors or missing messages. The reason is because this thesis is not focused on communication protocols.*

Assumption 7.2.3 *Each task has a unique id and can only be performed by one worker at the same time.*

Assumption 7.2.4 *All previous data used before a register has been stored into the worker's LB, is error-free. That means, all information used and performed by a task prior to a register stored in the worker's LB is error-free.*

Assumption 7.2.4 is related to the detection method. So, we are assuming that there is a good detection mechanism allowing that the information stored in the worker's LB is error-free, because if a fault affects a worker, the worker immediately performs the restore mechanism and the LB does not contain any register affected by the fault.

Assumption 7.2.5 *A worker can access only its own LB and cannot access the LB of other workers. Additionally, the leader cannot access a worker's LB.*

Assumption 7.2.6 *There is no direct communication between workers. A task communicates with other tasks through streams.*

Assumption 7.2.7 *The information stored into a LB is always just 0 or 1 step ahead of the real behaviour of the application performed.*

Theorem 7.2.8 *The log-based rollback-recovery algorithm proposed does not generate orphan messages and accomplishes the always-no-orphans consistency condition [6]:*

$$\forall e.\ \neg Stable(e) \implies Depend(e) \subseteq Log(e)$$

Proof 7.2.8 We need to prove that every non-deterministic event is not lost after performing the restore mechanism. To prove this, we checked what happens if a fault is detected between the register written and the action taken, and if no orphan messages were created.

A *START* register stores the checkpoint of the task before performing the task. If a failure appears after storing the register, the restore mechanism restarts the task and the worker undertakes the task under the restore mechanism's supervision . $\boxed{\text{Result: the } START \text{ register does not generate orphan messages}}$

A *READ* register has stored a copy of the data read from a stream. This register is stored before the task performs the data. Let's assume that a fault appears after storing the register, and before the task performs the data and removes the data from the stream. The fault is detected and the restore mechanism is called. Then the task is restarted until performing this register again. In this second time performing the register using the restore mechanism, the task reads the data stored into the register and the worker sends a confirmation message that the task has read the data from the LB to the FTC. The FTC uses the message received from the worker to search if the data has been removed in the input stream. Therefore, if the data read by the task is still into the input stream, then the FTC removes this data to the input stream. Otherwise, the FTC does nothing. $\boxed{\text{Result: there is no orphan message created}}$

A *WRITE* register stores a copy of the output data before the task writes it to a stream. Following the same technique, we assume the appearance of a fault after storing the register and before the task writes the data to the stream. The worker asks FTC if the data were written by the task before detecting the fault. FTC checks the leader's stable storage by looking for a task-related *STREAM* register with the data the worker asks for. If there is a register, FTC replies to the worker that the data were written before, otherwise, FTC answers that the data were not written before, and the task writes the data to the stream and the worker sends a copy of the data to FTC.

Result: No orphan messages are created

A *BLOCKED* register allows blocking the LPEL feature to the algorithm. Let's imagine that a failure appears after storing the register and before the task is returned to the leader. When the restore mechanism reads the *BLOCKED* register, the *IsTaskReceived* parameter is *FALSE*. So the restore mechanism forces the worker to send the task to the leader. After sending the task to the leader, the restore mechanism finishes and the worker awaits the confirmation message from the FTC.

Result: No orphan messages are created by the *BLOCKED* register

The *END* register offers two possibilities: On the one hand, a fault can appear after a task has finished, but before the worker stores an *END* register and returns the task to the leader. When that happens, the task is restarted by our restore mechanism using the other registers stored in the worker's LB. In this second round, once the task is finished, the *END* register is created and the task is returned to the leader.

On the other hand, a fault can appear after storing the register and before the task is returned to the leader. The restore mechanism reads the *END* and checks the *IsTaskReceived* parameter. In this case, *IsTaskReceived* is *FALSE* because the task was never sent to the leader and FTC never sent a confirmation message. Then the restore mechanism forces the worker to return the task to the leader. As a result, the restore mechanism finishes and the worker awaits the confirmation message from FTC.

Result: No orphan message is created after performing the restore mechanism.

In conclusion, our protocol satisfies the always-no-orphans condition. □

Theorem 7.2.9 *The recovery of a process is consistent by assuming a reliable LB and is free from the domino effect.*

Proof 7.2.9 The stored register ensures that a non-deterministic event is unified in the correct sequence in the workers' LB. Upon recovery, the latest checkpoint of the task becomes available at the *START* register in the workers' LB. The worker uses the registers stored in the LB to reconstruct the state of the task.

As the proof in Theorem 7.2.8, all the registers are used to restore the state of the worker and no orphan messages are generated. Therefore, recovery of the worker is consistent if assuming a reliable worker's LB. Furthermore, no other worker is required to rollback. Thus the domino effect of the unbounded rollback propagation is not possible. □

Theorem 7.2.10 *If a worker is concurrently affected by multiple failures, then the worker is restored to the status before the first fault appeared.*

Proof 7.2.10 If a fault affects a worker, then the restore mechanism is performed by the worker restoring the status of the task before the fault appears. So the task is restarted by the first step of the restore mechanism. Suddenly, the worker is again affected by another fault while performing the task with the restore mechanism. In this case, the restore mechanism is restarted. Execution of the restore mechanism does not affect relevant registers in the worker's LB because the restore mechanism never removes a register from the worker's LB.

Therefore, this second execution is performed as if the first restore mechanism had never happened. As a result, multiple concurrent faults in a worker are handled by restarting the restore mechanism without it affecting the registers stored in the worker's LB of the worker. So the worker is restored to the state that existed before the first fault appears. □

Theorem 7.2.11 *Only the registers that are not used for future recoveries by the restore mechanism are removed.*

$$(\forall r.\exists t.\neg \mathit{useful}(r,t)) \implies (\exists t_1.(t_1 \geq t) \wedge \mathit{removed}(r,t_1))$$

Proof 7.2.11 A register is removed only by garbage collection because its precondition for being removed is accomplished. Then we need to prove if each register is correctly removed to stable storage.

- *START*: Let's assume that the register is removed before storing the *END* register, and that a fault appears before the worker finishes the task. The restore mechanism cannot restart the task because the checkpoint is no longer in the LB and the worker cannot recovery the status before the fault appeared. If a task was finished by a worker, restarting the task is useless. The reason why is, given Assumption 7.2.4, the task was performed with no faults and, therefore it was not necessary to perform it again.

  Result: the *START* register can be removed only after storing the *END* register related to the same task with *IsTaskReceived* as *TRUE* or *FALSE*, or if FTC informs that the task was completed by another worker.

- *READ*: Let's assume that the register is removed before storing the *END* register, and that a fault appears before the worker finishes the task. When the task is performed by the restore mechanism, this task attempts to find a *READ* register related to the data in the worker's LB. The register is removed. So this register is no longer in the LB and the task tries to read the required data from the stream. However, these data were already removed from the stream before the fault occurred. So the task cannot continue its execution, which means that the worker cannot recover the status before the fault appeared. This register only becomes useless when the task is completed.

Result: the *READ* register has the same precondition as the *START* register.

- *WRITE*: Let's imagine that garbage collection removes a *WRITE* register before the precondition described. When the task is performed a second time by the worker with the restore mechanism, the task finds no impediment to rewrite the data to the stream. This implied a duplicated data problem: the data were written twice, first before the fault appeared and then after restarting the task. The task needs to be completed to avoid this problem. Therefore, one condition is to wait until the *END* register is stored, regardless of the *IsTaskReceived* parameter being *TRUE* or *FALSE*.

  Now let's assume that a fault appears when the *WRITE* register is stored and before writing the data. If the task is restarted, then the data are not written to the stream, but are removed when storing the *END*. The data were never written to the stream and the register with a copy of the data was removed when the register was still useful. Therefore, the *WRITE* register needs confirmation that the data were written. For this reason, this register awaits a confirmation message from FTC. So another condition of the *WRITE* register is to receive message from FTC confirming that the data were correctly written.

  Result: the *WRITE* register becomes useless after storing the *END* register and the worker receives a message from FTC confirming that the data were stored in the stream

- *BLOCKED*: One contradiction is that we can assume that the *BLOCKED* register is removed before becoming useless. Then the worker where the *BLOCKED* register was prematurely removed is affected by a fault. So the task stored in the worker's *START* register is restarted. Therefore, the task is performed by the worker, while the original task is either performed by another worker or is in the leader's task queue awaiting to be performed by a worker. This problem creates a duplication task in the system, so the *BLOCKED* register can be removed when the task finishes.

  Additionally, if a *BLOCKED* register is removed, but registers related to the task remain in the LB, then the worker that uses the restore mechanism can restart the task and the duplication task problem appears. The *BLOCKED* register needs to wait until the *START*, *READ* and *WRITE* registers of the task are removed from the LB.

Result: the *BLOCKED* register becomes useless when the task iss finished by this worker or by another worker, and also when there are no more registers related to the same task in the LB where the register is stored, except other *BLOCKED* registers and the *END* register

- *END*: Let's assume that the *END* register is removed when it is still useful. The restore mechanism restarts the task and is performed again. The worker is busy performing the task that was previously performed correctly. The worker can send the same task to leader twice. So the worker needs to know if the task was sent to the leader before removing the *END* register. Therefore, the *END* register becomes useless when the worker receives a message from FTC confirming that the leader has received the task.

  Additionally, if the *END* register is removed when there is another register related to the task in the same LB, the worker can restart the task again. Here a duplication task problem can appear. To avoid this problem, the *END* register needs to be the last register related to the task that is removed. Therefore, all the other task-related registers need to be removed from the worker's LB.

  Result: the *END* register becomes useless when the worker receives a message from FTC confirming that the task has arrived with no problems and when are no more registers related to the task in the LB where the register is stored.

- *SENT*: If this register is removed before becoming useless, FTC does not know to which workers to send the relevant registers that another worker asked for. This register becomes useless when the related task is finished by a worker and the leader receives the task. So the worker where the task was performed no longer asks FTC for relevant task data.

  Result: the *SENT* register is useless when the *RECEIVED* register is created with *IsFinished* parameter as *TRUE*.

- *RECEIVED*: Following the same strategy, a register *RECEIVED* with *IsFinished* as *FALSE* is removed before becoming useless.

  Let's imagine that the leader has crashed and a new leader is elected. Then FTC checks if the tasks with the *SENT* register and without the *RECEIVED* register are still in workers. As a result, FTC finds a *SENT* register, which was prematurely removed. In this case, FTC asks the worker if it performs the task, but the worker replies that it does not. So the leader tries to send the task to the worker. However, the task is perhaps not in the task queue and the leader cannot send it to the worker. Therefore, this register can be removed only when the *RECEIVED* register with the *IsFinished* parameter as *TRUE* is stored in the leader's stable storage.

  Result: this register has the same precondition as the *SENT* register.

Let's assume that the *IsFinished* parameter is *TRUE* for the *RECEIVED* register and that this register is removed from stable storage before time. When this register is created in the leader's stable storage, FTC sends a message to inform all the workers related to the task that the task has finished.

If the leader fails and a new leader is elected, FTC sees that there is a *SENT* register and no *RECEIVED* register. Therefore, FTC asks the worker if it performs the task and the worker replies that it does not have the task. Then the leader sends the task to the worker. This is similar to the *RECEIVED* register with the *IsFinished* parameter as *FALSE*. However, the solution is to keep it until all the other task-related registers have been removed. If a new leader is elected and FTC finds this register, then FTC calls the garbage collection mechanism. As a result, this register needs to wait until all the other task-related registers have been removed to avoid some registers not being removed if the leader crashes.

Result: this register can be removed by garbage collection when the leader's stable storage has no more task-related registers.

- *STREAM*: Last but not least, let's consider that a *STREAM* register is removed before becoming useless. This register needs to be in the leader's stable storage until the worker with the *WRITE* register finishes the task. If it is removed before the worker finishes and the worker starts the restore mechanism, then FTC thinks that these data were never written to the stream. As a result, this register has to be in the stable storage until the worker finishes the task.

  Result: the *STREAM* register becomes useless when a message from a worker confirms that the related *WRITE* register has been removed from the worker's LB.

  Let's imagine that this register is removed before FTC confirms that the data in the stream were correctly written. Thus there is a worker with a *WRITE* register that was never removed from the worker's LB. Therefore, this register waits until FTC compares the data in the stream to the data in this register.

  Result: the *STREAM* register can be removed when the *RECEIVED* register with *IsFinished* as *TRUE* is stored in the leader's stable storage, and FTC uses this register to compare the data stored in the register to the data stored in the stream.

□

7.3 Chapter Summary

This chapter presents the definitions, lemmas and proofs to demonstrate that the leader election and log-based rollback-recovery proposed accomplished their fault tolerance objectives.

The first part of this chapter is focused on the leader election algorithm proposed. Lemma 7.1.5 shows that Algorithm 6 cannot guarantee that only one leader is elected after the leader election mechanism is completed. For this reason, Lemma 7.1.5 has been replaced with Lemma 7.1.6, to demonstrate that Algorithm 7 assures that only one node can be elected as a leader. In the meanwhile, the rest of nodes update their local information with the information stored in the shared memory.

Furthermore, Lemma 7.1.10 confirms that Algorithm 7 has a bound of execution steps to be completed, what confirms the algorithm proposed is wait-free and lock-free.

The second part of this chapter is focused on our log-based rollback-recovery algorithm. A good log-based rollback-recovery must to ensure that a consistent state is accomplished after a fault is detected and no orphan messages are created. It is demonstrated by Theorem 7.2.8 and Theorem 7.2.9 that the log-based rollback-recovery algorithm proposed in this thesis accomplishes the always-no-orphan condition and ensures that a consistent state is accomplished in case of a fault affects a worker.

Last but not least, after performing our garbage collection mechanism is ensured that the nodes' stable storage do not contain obsolete registers. Theorem 7.2.11 corroborates the preconditions defined in Section 6.1.4 to remove obsolete registers into the nodes' stable storage.

Chapter 8

Assessment

This chapter evaluates the performance of a LPEL [109, 101] prototype that uses our leader election and the log-based rollback-recovery mechanism in experiments.

The hardware used for testing the experiments is a cluster property of the University of Hertfordshire. This cluster is a 144-node Linux cluster that contains a SMP machine with 48 cores and 256 GB Ram, and used by members of the Science and Technology Research Institute.

In this section, the execution time overhead represents the extra percentage of the execution time, in seconds, that the system needs to finish a program when a mechanism is called.

8.1 Evaluation of Leader Election

8.1.1 Description of the Experiment

The leader election experiment is based on sending and receiving tasks continuously between the workers and the leader. Tasks used for the experiment have a small execution time and they are insignificant for the performance metrics showed in the results.

This experiment forces the workers to keep contact with the leader, as well as the workers can detect a failure of the leader using the notification message method described in Section 4.2. Therefore, in case the leader fails, workers will follow the notification message behaviour explained in Figure 4.1.

Figure 8.1 shows the behaviour of the leader during the experiments performed.

At the beginning of each experiment, the system is forced to select core 0 as a leader, the remaining cores are workers. When each test starts, all the workers send a request task to the leader and then the leader replies with a task for each worker. Once the leader has replied all workers, it will wait until it receives a task from a worker. When the leader receives a task from a worker, the leader chooses randomly if it replies the worker and continues being the leader or it destroys itself, being reintegrated as a worker in the system. If the

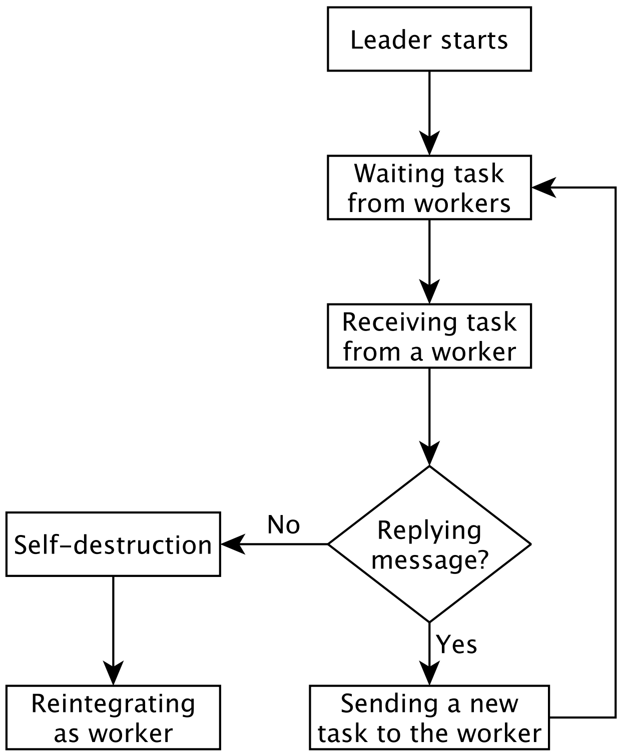

Leader starts
Waiting task from workers
Receiving task from a worker
Replying message?
No
Yes
Self-destruction
Reintegrating as worker
Sending a new task to the worker

Figure 8.1. Behaviour of the leader into the experiments

leader destroys itself, the system considers the leader fails, then the workers can start the leader election using the notification message method.

Using the cluster of University of Hertfordshire that has 48 cores, the experiment has been tested for 4, 8, 16, 32 and 48 cores. All the cores participate in the experiments, if 4, there will be 3 workers and 1 leader; if 8, there will be 7 workers and 1 leader; and so on. Additionally, for each group of cores, the experiment has been performed 50 times.

8.1.2 Discussion of Results

Figure 8.2 presents the amount of time used by all the workers to elect a new leader, as well as the time used by the remaining workers to know that the leader has changed. The figure shows a comparison of the execution time of every test realized for each groups of cores.

The average execution time of each group of cores in the figure are:

- 4 cores: 0.1284 seconds
- 8 cores: 0.215 seconds
- 16 cores: 0.3708 seconds
- 32 cores: 0.8662 seconds

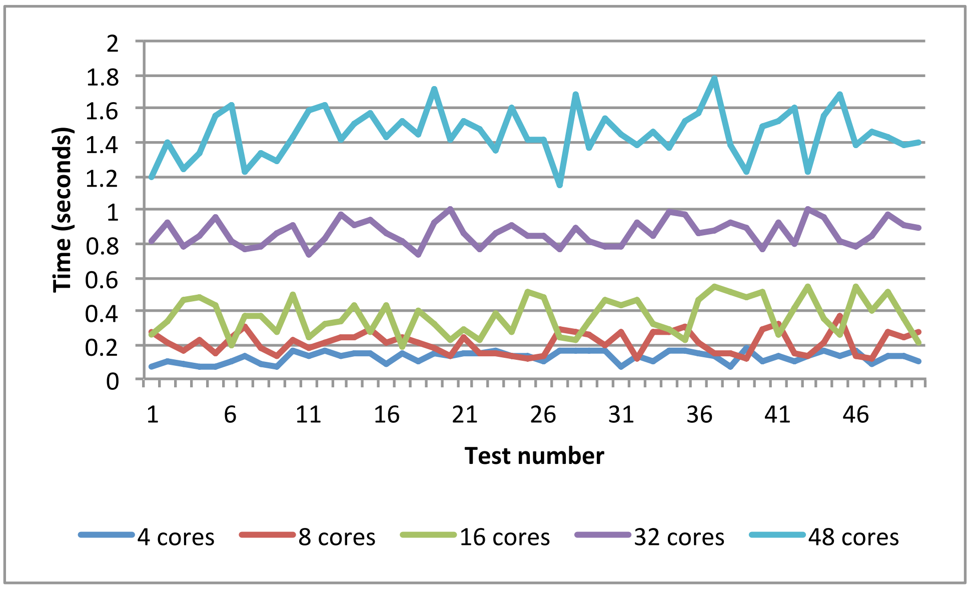


Figure 8.2. Time (sec) used by the nodes to elect a new leader

- 48 cores: 1.4548 seconds

As we can see, time needed for workers to know that the leader has changed increases with the number of workers, as usual in this kind of leader election mechanisms. The reason is because there are more number of workers that need to update their local information with the new leader information.

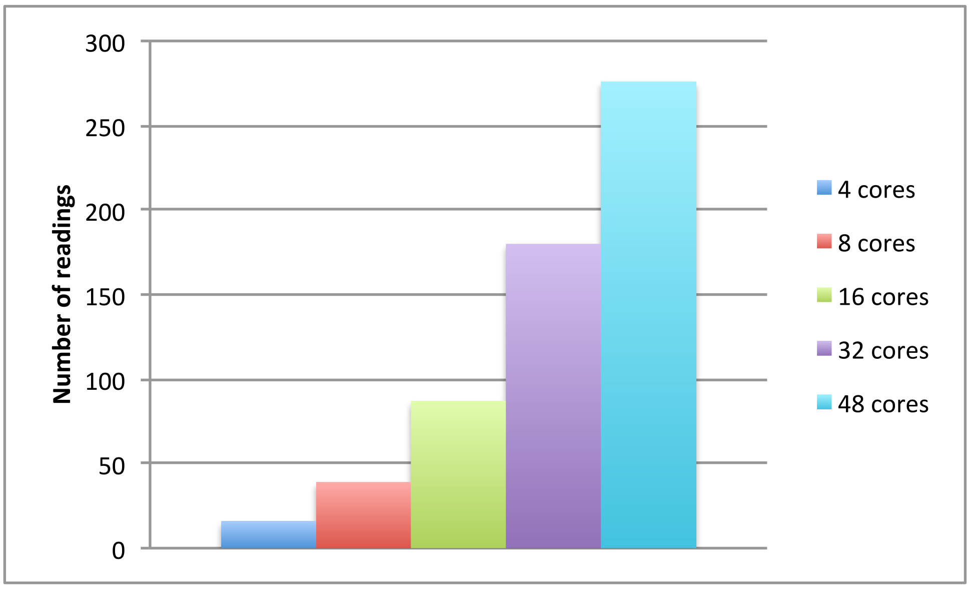


Figure 8.3. Number of times all the workers did a reading from the shared memory

Figure 8.3 shows the average number of times all the workers read from the shared memory per leader election. The figure also represents the amount of readings from the shared memory per group of tested cores. The candidates in our leader election mechanism always write 2 or 3 times to the shared memory, so the number of writing to the shared memory to elect a leader is constant irrespectively of the participants.

The average number of times per group of cores tested 50 times to elect a leader in the Figure 8.3 are:

- 4 cores: 16 readings
- 8 cores: 39 readings
- 16 cores: 87 readings
- 32 cores: 180 readings
- 48 cores: 276 readings

In consequence, Figure 8.3 shows that the result of the evaluation in Section 5.6.3 is correct, since that, in the worst case, the workers reads 6*(n-1) times when the leader election starts.

In some distributed systems with shared memory, reading data from the shared memory can have a high execution time, for this reason reducing the number of readings is important. The figure shows how the amount of readings to the shared memory depends on the number of participants, given that nodes need to update their local information with the information stored in the shared memory about the new leader. Accordingly, our algorithm has been developed in order to save time, keeping low the number of readings to the shared memory.

Additionally, our algorithm has a constant number of writings independently of the number of participants, what allows it use with any number of nodes without increasing the amount of writings.

Figure 8.4 compares the amount of reading that the nodes need to realize, either shared memory readings or messages related to the leader election. In this figure, our leader election approach is compared to other leader election explained in Section 3, such as, RatRace, RMR and Enhanced Bully Algorithm. Rat RatRace, RMR and our approach do not have reading from messages, instead of Enhanced Bully Algorithm only the leader reads and writes from the shared memory, but the nodes read and write messages from other nodes. As a result, our leader election is the mechanism that the nodes make less readings from the shared memory.

This figure shows the execution time overhead that our leader election gives to system. As we can see, the overhead of our algorithm is linear. Increasing the number of nodes in the system increases execution time that all nodes need to know that the leader has changed.

Figure 8.6 compares the execution time overhead of our approach (Leader Election) to other approaches, such as, RatRace, RMR and Enhanced Bully Algorithm.

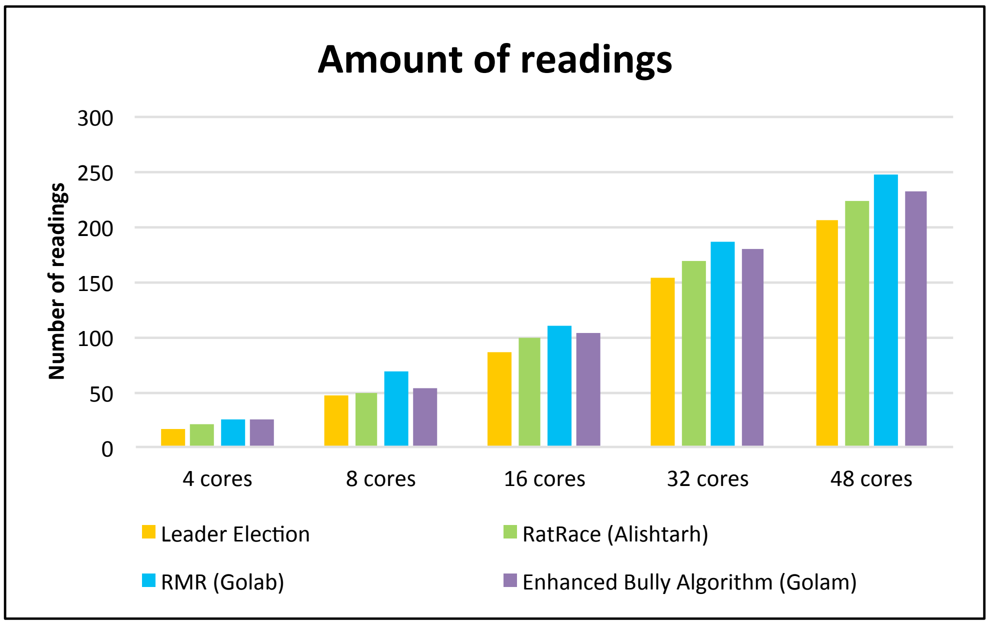


Figure 8.4. Number of reading comparative of different approaches including ours leader election approach.

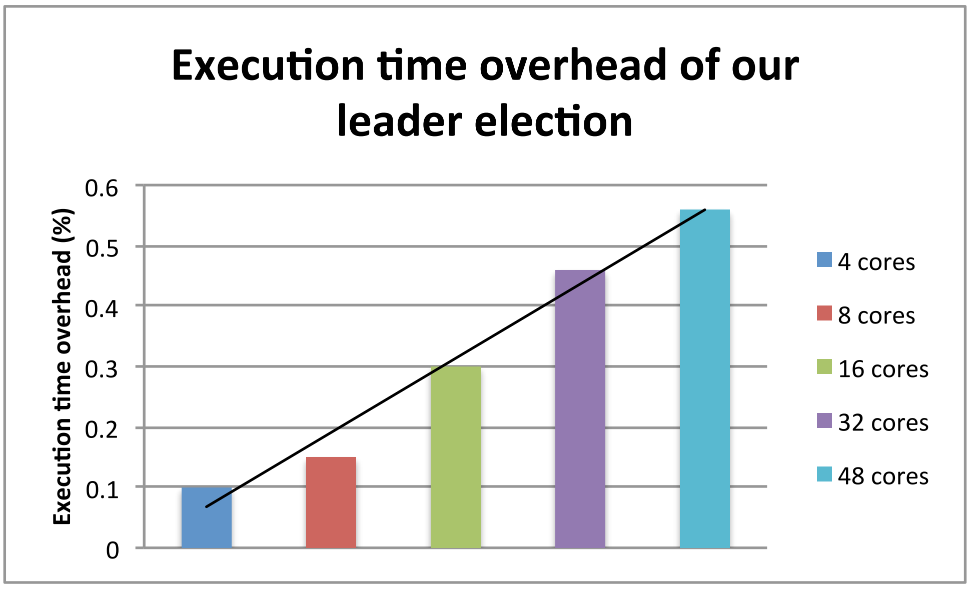


Figure 8.5. Execution time overhead of our leader election approach.

Our algorithm was developed in order to save time, keeping low overhead. With this figure we can say that we accomplish the goal to create an efficient leader election that has a low execution time overhead.

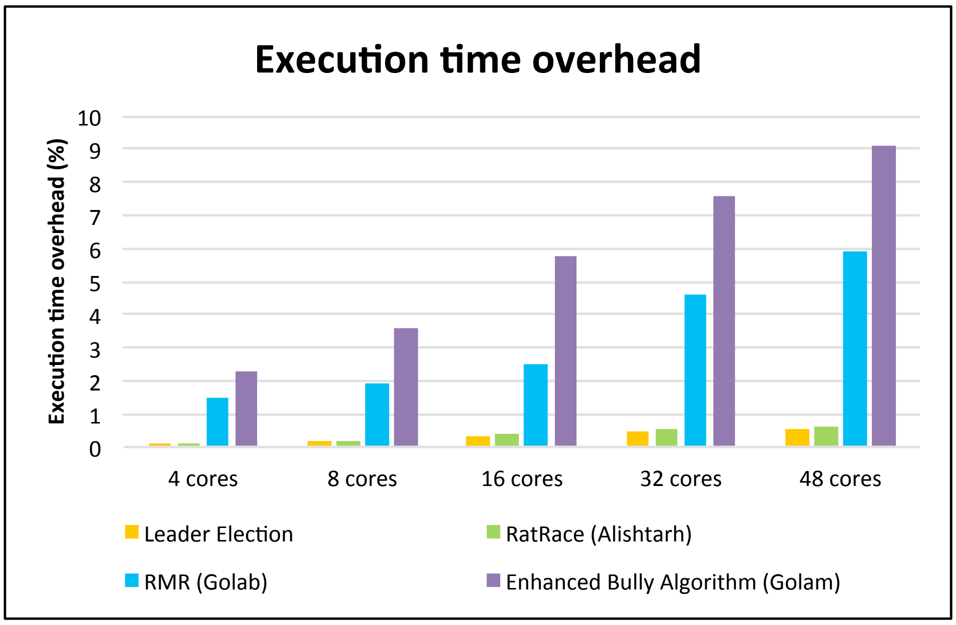


Figure 8.6. Execution time overhead comparative of different approaches including ours leader election approach.

8.2 Log-based rollback-recovery

8.2.1 Description of the Experiments

A specific kind of tasks have been created for the log-based rollback-recovery experiments with the purpose they are able to read and write data continuously to streams. In these experiments, each task has been developed for reading from the stream 300 times and writing to the stream 500 times. These experiments were also performed for 4, 8, 16, 32 and 48 cores.

On account of this thesis is not focused on creating a fault detection method, to simulate the experiments we injected faults into the workers. For this reason, when a fault is injected into a worker, the worker calls the restore mechanism.

Different behaviours have been tested, the most representatives can be summarize in:

1. Not using our algorithm and without faults: tests perform without our log-based rollback-recovery algorithm. This is the basic model, the other experiments are based and compare on this first behaviour case.

2. Using our algorithm and without faults: tests performed base on the basic model, previous example, but using our log-based rollback-recovery algorithm. Workers are not affected by faults.

3. Using our algorithm and with one fault: tests performed are an alteration of the second behaviour (using our algorithm and without faults) but there

is one worker affected by one fault. The remaining workers are not affected by the fault.

4. Using our algorithm and with several faults: Following the behaviour 3 but more than one fault occurred. Each time a task reads or writes to the stream, the worker performing the task can be affected by a fault. The probability of a fault can affect a worker each time a task reads or writes to the stream, is 1 in 1000.

8.2.2 Discussion of Results

Figure 8.7 presents the execution time overhead caused by the restore mechanism using the registers stored in the LB of the worker. The following list shows the possibilities of the registers stored in the LB of the worker when the restore mechanism is called, as well as the execution time overhead that the workers receives after it is restarted by the restore mechanism.

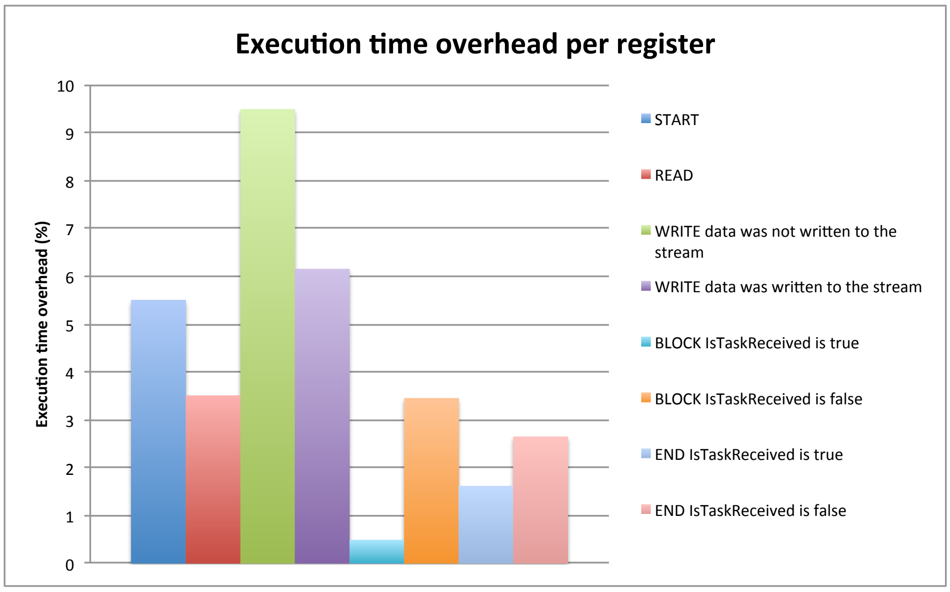


Figure 8.7. Percentage of the execution time overhead depending on the registers stored in the stable storage

- START register: 5.451
- READ register: 3.512
- WRITE register without data written to the stream: 9.489
- WRITE register with data written to the stream: 6.131
- BLOCK with *IsTaskReceived* parameter as TRUE: 0.493

- BLOCK with *IsTaskReceived* parameter as FALSE: 3.45
- END with *IsTaskReceived* parameter as TRUE: 1.623
- END with *IsTaskReceived* parameter as FALSE: 2.65

The execution time overhead is less than 4%, except for the WRITE and START registers. The overhead of a START register is because of the restore mechanism interrupts the execution of the worker and forces the worker to restart the task using the checkpoint stored in the START register. The WRITE registers have the highest execution time overhead. This is due to the worker checks with FTC if the data has been written by the task before the fault was detected. If the data was not written before, the overhead is highest, since the task has to write the data into the stream.

Through all these experiments, we can conclude that our restore mechanism gives to a worker a low execution time overhead per register stored into the LB of the worker.

Additionally, these practical experiments have demonstrated that the worker can achieve a consistent state when the restore mechanism is called, avoiding orphan messages and not modifying the output of the application.

Figure 8.8 illustrates the execution time overhead caused by our log-based rollback-recovery algorithm. First of all, We obtained the time (t_e) to perform the experiments for each group of cores without using our algorithm (behaviour 1). Given that its value is 0, it is not represented in the figure. Secondly, we obtained the time (t_f) of the experiments with the proposed log-based rollback-recovery without injecting faults in the system (behaviour 2). Comparing behaviour 1 and 2, the execution time overhead is represented as $(t_f * 100)/t_e$, where the excess of 100% is the overhead caused by our algorithm. Additionally, in the figure is also represented the execution time overhead caused by our approach in case a fault only affects one of the workers (behaviour 3), or if several faults appear in different workers (behaviour 4).

In order to complete the log-based rollback-recovery experiments, we also checked the overhead caused by our restore mechanism performing each register stored into the LB of the worker.

As a result, we can conclude that the average execution time overhead caused by our algorithm is:

- Without faults (Behaviour 2): 8.75%
- One fault affects only one of the workers (Behaviour 3): 12.44%
- Several faults affecting different workers (Behaviour 4): 24.92%

As the figure shows, our log-based rollback-recovery has a constant overhead when workers are not affected by faults (Behaviour 2), regardless of the number of workers in the system. For the experiments with one fault (Behaviour 3), the figure shows how the approach also has a constant overhead, irrespectively of the number of workers.

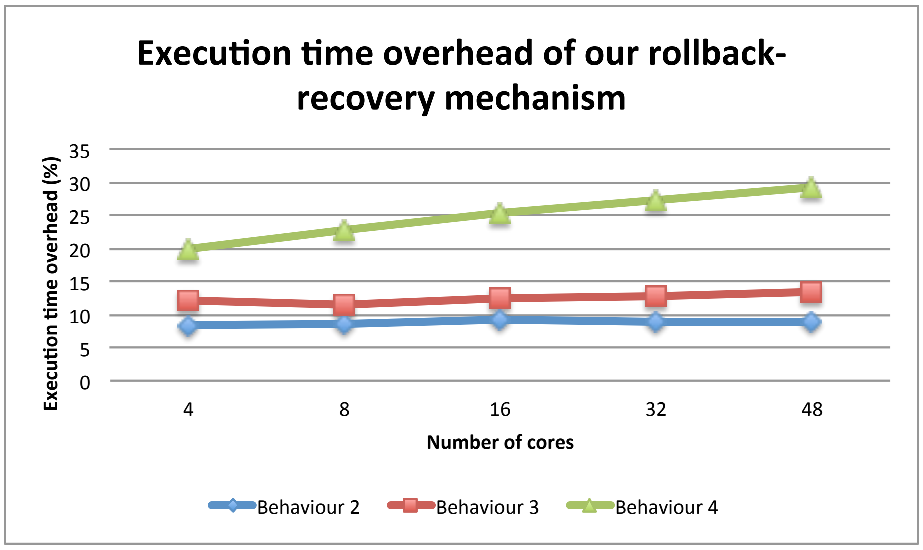


Figure 8.8. Percentage of execution time overhead caused by our log-based rollback-recovery algorithm

With respect to the experiments with several faults, the number of tasks increase according to the number of workers used by the system. The more workers are performing tasks, the more probability of an apparition of a fault will can be. For this reason, the experiments with 48 cores have more execution time overhead than the experiments with less cores.

In conclusion, we can assert that our algorithm is scalable, due to the number of workers does not affect the execution time overhead when our algorithm is utilised in error-free.

Other of the objectives of our proposal was to create a log-based rollback-recovery algorithm that could be performed keeping a low execution time overhead in error-free. For this reason, the garbage collection mechanism is being performed continuously by our log-based rollback-recovery algorithm, removing the obsolete registers into the stable storage of the nodes, Based on the data exposed in Figure 8.8, we can affirm we have accomplished this goal, since our garbage collection mechanism minimises the execution time overhead.

Our restore mechanism is performed in the worker affected, avoiding the interruption of other workers or leader, achieving a consistent state after the restore mechanism is called. This is due to our algorithm is asynchronous, conferring this feature to the workers performance in case a fault appears. For this reason, in Figure 8.8, the results of Behaviour 3 are constant, given that the fault only affects one worker while the remaining workers can continue with their execution. The results shows how the restore mechanism avoids the domino effect.

Figure 8.9 shows the execution time overhead when the system is not affected by faults comparing our proposed mechanism to other rollback-recovery mechanism explained in Section 3, such as, MPICH-V, Mutable Checkpoints and Chaoguang. One of the objectives was to create a rollback-recovery that

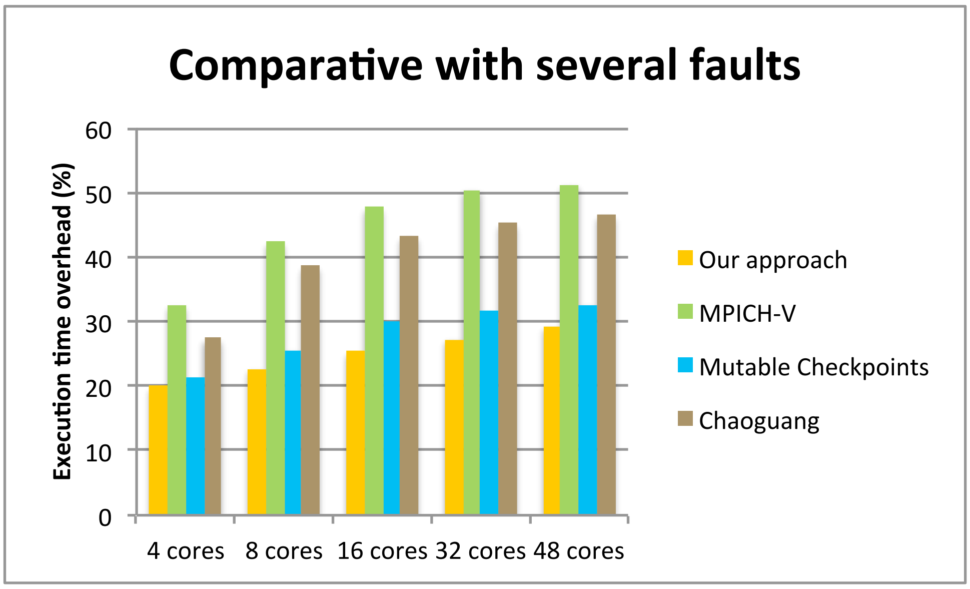


Figure 8.9. Percentage of execution time overhead when the system is not affected by faults comparing other approaches to our log-based rollback-recovery approach

can keep a low execution time overhead. Our algorithm accomplishes this goal thanks to the creation of the garbage collection mechanism and the implementation of the restore mechanism using the registers stored, we create and efficient rollback-recovery mechanism that minimizes the execution time overhead.

Figure 8.9 compares algorithms without faults, that means, in error-free time. Figure 8.10 compares same algorithms when the system is affected by several faults. Our algorithm is still the algorithm with less execution time overhead, because our restore mechanism is performed only by the affected worker, avoiding interruption and synchronization of other workers, achieving a consistent state after the restore mechanism is called. For example, MPICH-V has the higher overhead because the nodes can be affected by the domino effect. As it is demonstrated in Theorem 7.2.9 (Section 7.2), our algorithm is free from domino effect.

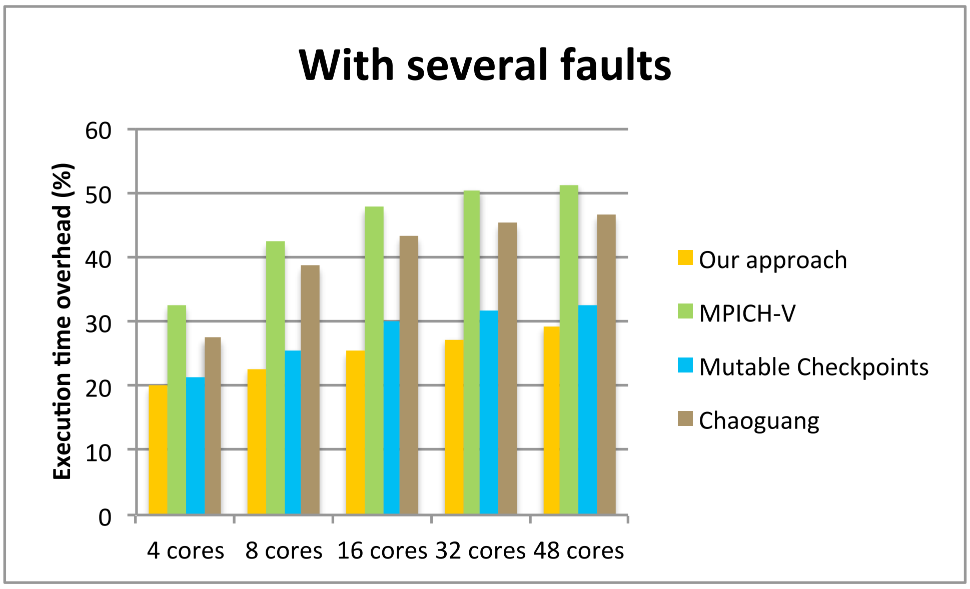


Figure 8.10. Percentage of execution time overhead when the system is not affected by faults comparing other approaches to our log-based rollback-recovery approach

Chapter 9

Conclusion

This chapter summarizes the main features and contribution of this thesis. It also contains an overview of possible future research directions in fault tolerance in distributed systems with shared memory.

9.1 Thesis Summary

As it is shown in the related work (Section 3), there are not too many works oriented to create a fault tolerance mechanism for distributed system with stream processing networks. Therefore, we developed two fault tolerance mechanisms, leader election and log-based rollback-recovery, that work with stream processing networks. Then the combination of the leader election and the log-based rollback-recovery proposed are optimised, using the properties of the stream programs, such as, asynchronous property.

These two fault tolerance mechanism are being implemented as a prototype using LPEL. These fault tolerance mechanisms help increase the dependability of applications, and can handle permanent and transient faults that affect the system's hardware.

Both presented mechanisms have been optimised for distributed systems with shared memory, a paradigm that is becoming increasingly important because many-core processors feature *network on chip* (NoC).

The first mechanism that we have presented is for leader election in systems that depend on a central leader process. Such processes are often used to manage shared resources, or to distribute work over several worker processes. They are, therefore, crucial for correct system operation. If the leader process fails, a new leader process is elected from a collection of candidate processes.

In our research, we found only a few leader election approaches for shared memory [130, 120, 45]. Unlike them, Chapter 5 proves that our leader election method has less space complexity and a faster response time. As proved in Section 7.1, our algorithm has a constant space complexity per node $O(2)$ and has a constant time complexity per node. However, the algorithm's time complexity with all number of cores is linear $O(n) = [6 * (n - 1)] * T_{\text{read}} + 3 * T_{\text{write}}$ for the worst case, where n is the number of nodes used in the system.

Additionally, the communication protocol and the topology of the network related to the system are not a restriction for our leader election algorithm. Thus they are not decisive elements in our proposal, which facilitates the application of our algorithm to other distributed systems.

Regarding the second mechanism, log-based rollback-recovery is based on taking snapshots of the state of a task, and offers the possibility of recovering a previous state should the worker fail where the task is performed. The suggested restore mechanism uses the information stored in the stable storage to rollback the task to the previous state found before the fault appeared.

Furthermore, specific garbage collection for the algorithm is created for the algorithm and is presented in Section 2.1.5. In is unlike other approaches [23, 108] where the recovery strategy is presented, but garbage collection is never used. Thus the presented garbage collection can be used for other rollback-recovery strategies for distributed systems.

In summary, our log-based rollback-recovery approach has the following advantages respect with other algorithm developed for the same task (presented in Section 3):

1. It is a domino effect-free algorithm
2. The experimental result in Chapter 8 shows that the proposed log-based rollback-recovery has a lower overhead and that no faults appeared.
3. The recovery overhead depends on how many faults appear in the system.
4. A global consistent state after a worker is restored is ensured.
5. It is scalable since log-based rollback-recovery is insensitive to the number of workers in the system.

Besides, the Compiler Technology and Computer Architecture Group (CTCA) of the University of Hertfordshire is an important member in the CRAFTERS [116] project and contributes to work for compiler-supported fault tolerance to allow the design of robust concurrent systems with manageable complexity to confer LPEL and S-Net in order to achieve this purpose.

I am a member of the CRAFTERS project and a researcher at the in University of Hertfordshire. Therefore, given the combination of these two mechanisms and their implementation into LPEL, consequently S-Net has contributed to create a strong compiler-supported fault tolerance which the CRAFTERS project requires to achieve its objective.

9.2 Future work

The work conducted in this thesis provides some basis for future research lines. These research lines can be:

- Stateful components

- Dynamic changes of the network structure
- Granularity of the logging approach
- Adding additional fault tolerance mechanisms to the proposed LPEL prototype
- Creating a fault detector
- Correctness of log-based rollback-recovery

Stateful components

The log-based rollback-recovery approach presented in this thesis only aggregates messages and it does not value computations to produce the result. The research in this thesis about log-based rollback-recovery was conducted with the objective to create an efficient mechanism to solve the transient and permanent faults that can affect workers of the system. So, the experimental approach created is adapted to LPEL which does not have stateful components. However, S-Net that uses LPEL to communicate with the hardware has stateful components. As a result, the approach should be modified to integrate stateful components that allow the full recuperation of S-Net in case of being affected by a transient or permanent fault; allowing the utilization of stateful components to produce the application's output.

Dynamic changes of the network structure

The system network topology is not a restriction of our approaches, leader election and log-based rollback-recovery. Nevertheless, both approaches presented do not have a mechanism to support dynamic changes of the network structure. Therefore, it would be a good idea to investigate and add a mechanism to the approaches presented in this thesis, allowing the adaptation of leader election and log-based rollback-recovery when the network topology changes during the application performance.

Granularity of the logging approach

The log-based rollback-recovery presented in this thesis works on all workers of the system without synchronisation among the workers. Besides, the approach presented cannot work correctly if the system is separated in clusters, because the approaches see all workers in only one cluster. Consequently, a modification of the approach could allow the user to select a group of workers to activate log-based rollback-recovery or participate in the leader election.

Adding additional fault tolerance mechanisms to the proposed LPEL prototype

There are a few fault tolerance techniques that can be added to this prototype as well.

The paper [80] discusses different fault tolerance techniques that can be added to S-Net and presents also three fault tolerance mechanisms:

- Checkpointing/restoring the program state
- Dynamic Reconfiguration
- Redundant Computation

In the present thesis, we are focused on the checkpoint/restore mechanism for log-based rollback-recovery. In the same way, the other two mechanisms proposed in the paper mentioned, might be an encouraging research direction. For example, these techniques could be focused on ensuring the communication protocol is error-free.

Creating a fault detector

Another objective of this thesis was the creation of two fault tolerance techniques for distributed systems with shared memory. In Chapter 8, the experiments presented use the notification message method to detect the leader's failure. In the log-based rollback-recovery experiments, a fault is injected into a worker and, at the same time, the worker is forced to call the restore mechanism to resolve the appearance of the fault. However, the thesis is focused on these two techniques and did not focus on fault detection to detect the appearance of faults in the system. In conclusion, no proper fault detector for both mechanisms exists.

In this case, a promising research line would be to create a fault detector that detects the fault and uses the correct fault tolerance technique to repair it. For example, a good start for a future research could be the used of a fault detection mechanism with signals models [67] as well as could be developed a fault detection mechanism with a process-identification method [69].

Bibliography

[1] A. Acharya and B. Badrinath. Checkpointing distributed applications on mobile computers. In *Parallel and Distributed Information Systems, 1994., Proceedings of the Third International Conference on*, pages 73–80, Sep 1994.

[2] Y. Afek, E. Gafni, J. Tromp, and P. M. Vitányi. Wait-free test-and-set. In *Distributed Algorithms*, pages 85–94. Springer, 1992.

[3] D. Alistarh and J. Aspnes. Sub-logarithmic test-and-set against a weak adversary. In *Distributed Computing*, pages 97–109. Springer, 2011.

[4] D. Alistarh, H. Attiya, S. Gilbert, A. Giurgiu, and R. Guerraoui. Fast randomized test-and-set and renaming. In *Distributed Computing*, pages 94–108. Springer, 2010.

[5] L. Alvisi, E. Elnozahy, S. Rao, S. Husain, and A. de Mel. An analysis of communication induced checkpointing. In *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, pages 242–249, June 1999.

[6] L. Alvisi and K. Marzullo. Message logging: Pessimistic, optimistic, causal and optimal. *IEEE Transactions on Software Engineering*, pages 149–159, 1998.

[7] L. Alvisi, S. Rao, and H. M. Vin. Low-overhead protocols for fault-tolerant file-sharing. In *In Proceedings of the IEEE 18 th International Conference on Distributed Computing Systems*, pages 452–461, 1998.

[8] A. F. Anta, E. Jiménez, and M. Raynal. Eventual leader election with weak assumptions on initial knowledge, communication reliability, and synchrony. *Journal of Computer Science and Technology*, 25(6):1267–1281, 2010.

[9] A. W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987.

[10] H. Attiya and J. Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.

[11] A. Avizienis and J. Kelly. Fault tolerance by design diversity: Concepts and experiments. *Computer*, 17(8):67–80, Aug 1984.

[12] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan. - Mar. 2004.

[13] D. Avresky and D. Kaeli. *Fault-Tolerant Parallel and Distributed Systems.* Springer US, 2012.

[14] O. Babaoglu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. *Distributed Systems*, 2:63–75, 1993.

[15] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. *ACM Transactions on Database Systems (TODS)*, 33(1):3, 2008.

[16] F. Bause and P. S. Kritzinger. *Stochastic Petri Nets.* Springer, 1996.

[17] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. *The Midway distributed shared memory system*. IEEE, 1993.

[18] B. Bhargava and S.-R. Lian. Independent checkpointing and concurrent rollback for recovery in distributed systems-an optimistic approach. In *Reliable Distributed Systems, 1988. Proceedings., Seventh Symposium on*, pages 3–12, Oct 1988.

[19] B. Bieker, E. Maehle, G. Deconinck, and J. Vounckx. Reconfiguration and checkpointing in massively parallel systems. In K. Echtle, D. K. Hammer, and D. Powell, editors, *EDCC*, volume 852 of *Lecture Notes in Computer Science*, pages 353–370. Springer, 1994.

[20] G. Birkhoff and S. Mac Lane. *Algebra*. AMS Chelsea Publishing, New York, 1999.

[21] P. E. Black. big-o notation. *Dictionary of Algorithms and Data Structures*, 2007.

[22] A. Borg, J. Baumbach, and S. Glazer. A message system supporting fault tolerance. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, SOSP '83, pages 90–99, New York, NY, USA, 1983. ACM.

[23] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. Mpich-v: Toward a scalable fault tolerant mpi for volatile nodes. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 29–29, Nov 2002.

[24] A. Bouteiller, T. Hrault, G. Krawezik, P. Lemarinier, and F. Cappello. Mpich-v project: A multiprotocol automatic fault-tolerant mpi. *IJHPCA*, 20(3):319–333, 2006.

[25] A. Bouteiller, P. Lemarinier, G. Krawezik, and F. Cappello. Coordinated checkpoint versus message log for fault tolerant mpi. In *CLUSTER*, pages 242–250. IEEE Computer Society, 2003.

[26] A. Bouteiller, T. Ropars, G. Bosilca, C. Morin, and J. Dongarra. Reasons for a pessimistic or optimistic message logging protocol in mpi uncoordinated failure, recovery. In *CLUSTER*, pages 1–9. IEEE, 2009.

[27] B. Bouteiller, F. Cappello, T. Herault, K. Krawezik, P. Lemarinier, and M. Magniette. Mpich-v2: a fault tolerant mpi for volatile nodes based on pessimistic sender based message logging. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 25–25, Nov 2003.

[28] F. V. Brasileiro, P. D. Ezhilchelvan, S. K. Shrivastava, N. Speirs, S. Tao, et al. Implementing fail-silent nodes for distributed systems. *Computers, IEEE Transactions on*, 45(11):1226–1238, 1996.

[29] A. Campos and M. Castillo. Checkpointing through garbage collection. In I. Acta Press, editor, *EDCC*, PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS;Parallel and distributed computing and systems. International conference; 8th, Parallel and distributed computing and systems. Springer, 1996.

[30] G. Cao and M. Singhal. Mutable checkpoints: a new checkpointing approach for mobile computing systems. *Parallel and Distributed Systems, IEEE Transactions on*, 12(2):157–172, Feb 2001.

[31] M. Carkci. *Dataflow and Reactive Programming Systems*. CreateSpace Independent Publishing Platform, 1 edition, 2014.

[32] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[33] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, Feb. 1985.

[34] K. M. Chandy and C. V. Ramamoorthy. Rollback and recovery strategies for computer programs. *Computers, IEEE Transactions on*, 100(6):546–556, 1972.

[35] G.-M. Chiu and C.-R. Young. Efficient rollback-recovery technique in distributed computing systems. *Parallel and Distributed Systems, IEEE Transactions on*, 7(6):565–577, Jun 1996.

[36] D. Clark. The design philosophy of the darpa internet protocols. In *Symposium Proceedings on Communications Architectures and Protocols*, SIGCOMM '88, pages 106–114, New York, NY, USA, 1988. ACM.

[37] I. Corporation. Paragon user's guide, 1993.

[38] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design.* Addison-Wesley Publishing Company, USA, 5th edition, 2011.

[39] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Proceedings KR'96*, pages 148–159. Morgan Kaufmann, 1996.

[40] F. Cristian. Understanding fault-tolerant distributed systems. *Commun. ACM*, 34(2):56–78, Feb. 1991.

[41] F. Cristian, S. Mishra, and Y. Hyun. Implementation and performance of a stable-storage service in unix. In *Reliable Distributed Systems, 1996. Proceedings., 15th Symposium on*, pages 86–95, Oct 1996.

[42] A. Derhab and N. Badache. A self-stabilizing leader election algorithm in highly dynamic ad hoc mobile networks. *Parallel and Distributed Systems, IEEE Transactions on*, 19(7):926–939, 2008.

[43] D. Dice, D. Hendler, and I. Mirsky. Lightweight contention management for efficient compare-and-swap operations. In *Euro-Par 2013 Parallel Processing - 19th International Conference,Aachen, Germany, August 26-30, 2013. Proceedings*, pages 595–606, 2013.

[44] A. Duarte. *RADIC: A Powerful Fault-tolerant Architecture.* Universitat Autònoma de Barcelona, 2007.

[45] M. Effatparvar, N. Yazdani, M. EffatParvar, A. Dadlani, and A. Khonsari. Improved algorithms for leader election in distributed systems. In *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, volume 2, pages V2–6–V2–10, April 2010.

[46] I. Egwutuoha, D. Levy, B. Selic, and S. Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326, 2013.

[47] E. Elnozahy and W. Zwaenepoel. Manetho: Transparent roll back-recovery with low overhead, limited rollback, and fast output commit. *IEEE Transactions on Computers*, 41(5):526–531, 1992.

[48] E. N. Elnozahy. On the relevance of communication costs of rollback-recovery protocols. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 74–79, New York, NY, USA, 1995. ACM.

[49] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Compututing Surveys*, 34(3):375–408, 2002.

[50] N. EPSRC. Archer is the latest uk national supercomputing service. the archer service started in november 2013 and is expected to run for 5 years.

[51] A. Fernandez, E. Jimenez, and M. Raynal. Electing an eventual leader in an asynchronous shared memory system. In *Dependable Systems and Networks, 2007. DSN '07. 37th Annual IEEE/IFIP International Conference on*, pages 399–408, June 2007.

[52] A. Fernandez, E. Jimnez, M. Raynal, and G. Trdan. A timing assumption and a t-resilient protocol for implementing an eventual leader service in asynchronous shared memory systems. In *ISORC*, pages 71–78. IEEE Computer Society, May 2007.

[53] L. Fialho. *Fault Tolerance Configuration for Uncoordinated Checkpoints*. PhD thesis, Univ. Autonoma de Barcelona, Spain, 2011.

[54] W. Fu and C. Hauser. A real-time garbage collection framework for embedded systems. In *Proceedings of the 2005 Workshop on Software and Compilers for Embedded Systems*, SCOPES '05, pages 20–26, New York, NY, USA, 2005. ACM.

[55] S. E. George, I.-R. Chen, and Y. Jin. Movement-based checkpointing and logging for recovery in mobile computing systems. In *Proceedings of the 5th ACM international workshop on Data engineering for wireless and mobile access*, pages 51–58. ACM, 2006.

[56] W. Golab, D. Hendler, and P. Woelfel. An \$o(1)\$ rmrs leader election algorithm. *SIAM J. Comput.*, 39(7):2726–2760, May 2010.

[57] W. M. Golab, V. Hadzilacos, D. Hendler, and P. Woelfel. Constant-rmr implementations of cas and other synchronization primitives using read and write operations. In I. Gupta and R. Wattenhofer, editors, *PODC*, pages 3–12. ACM, 2007.

[58] C. Grelck and F. Penczek. Implementing s-net a typed stream processing language part i compilation, code generation and deployment. Technical report, University of Hertfordshire, Hatfield, Herts, AL10 9AB, Dec. 2007.

[59] C. Grelck, S.-B. Scholz, , and A. Shafarenko. A gentle introduction to s-net: Typed stream processing and declarative coordination of asynchronous components. *Parallel Processing Letters*, 18(2):221–237, 2008.

[60] C. Grelck and S.-B. Scholz. Saca functional array language for efficient multi-threaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[61] C. Grelck, S.-B. Scholz, and A. Shafarenko. A gentle introduction to S-Net: Typed stream processing and declarative coordination of asynchronous components. *Parallel Processing Letters*, 18(2):221–237, 2008.

[62] C. Grelck and A. Shafarenko. Report on S-Net: A typed stream processing language, part I: Foundations, record types and networks. Technical report, University of Hertfordshire, Department of Computer Science, Compiler Technology and Computer Architecture Group, Hatfield, England, United Kingdom, 2006.

[63] R. Guerraoui and M. Raynal. A leader election protocol for eventually synchronous shared memory systems. In *Software Technologies for Future Embedded and Ubiquitous Systems, 2006 and the 2006 Second International Workshop on Collaborative Computing, Integration, and Assurance. SEUS 2006/WCCIA 2006. The Fourth IEEE Workshop on*, 2006.

[64] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems (2Nd Ed.)*, pages 97–145. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[65] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, Jan 1991.

[66] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, Nov 1993.

[67] R. Isermann. Fault detection with signal models. In *Fault-Diagnosis Systems*, pages 111–146. Springer Berlin Heidelberg, 2006.

[68] R. Isermann. *Fault-diagnosis systems : an introduction from fault detection to fault tolerance*. Springer, Berlin, 2006.

[69] R. Isermann. *Fault-diagnosis systems: an introduction from fault detection to fault tolerance*. Springer Science & Business Media, 2006.

[70] P. Jalote. *Fault tolerance in distributed systems*. Prentice-Hall, Inc., 1994.

[71] Q. Jiang, Y. Luo, and D. Manivannan. An optimistic checkpointing and message logging approach for consistent global checkpoint collection in distributed systems. *Journal of Parallel and Distributed Computing*, 68(12):1575–1589, 2008.

[72] D. B. Johnson. *Distributed System Fault Tolerance Using Message Logging and Checkpointing*. PhD thesis, Rice University Houston, Houston, TX, USA, 1990. AAI9110983.

[73] D. B. Johnson and W. Zwaenepoel. *Sender-based message logging*. Springer, 1987.

[74] D. B. Johnson and W. Zwaenepoel. Recovery in distributed systems using asynchronous message logging and checkpointing. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pages 171–181. ACM, 1988.

[75] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

[76] R. Jones and R. D. Lins. *Garbage collection: algorithms for automatic dynamic memory management*. Wiley, 1996.

[77] T.-Y. T. Juang and M.-C. Liu. An efficient asynchronous recovery algorithm in wireless mobile ad hoc networks. *Journal of Internet Technology*, 3(2):147–155, 2002.

[78] S. Kalaiselvi and V. Rajaraman. A survey of checkpointing algorithms for parallel and distributed computers. *Sadhana*, 25(5):489–510, 2000.

[79] W. H. Kersting. *Distribution system modeling and analysis*. CRC press, 2012.

[80] R. Kirner, V. S. Marco, M. Zolda, and F. Penczek. Fault-tolerant coordination of S-Net stream-processing networks. In *Proc. 2nd Workshop on Feedback-Directed Compiler Optimization for Multi-Core Architectures*, Berlin, Germany, Jan. 2013.

[81] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *Software Engineering, IEEE Transactions on*, SE-13(1):23–31, Jan 1987.

[82] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *Software Engineering, IEEE Transactions on*, SE-13(1):23–31, Jan 1987.

[83] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, and C. Senft. Distributed fault-tolerant real-time systems: The mars approach. *Micro, IEEE*, 9(1):25–40, 1989.

[84] A. D. Kshemkalyani and M. Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[85] J. G. Kuhl and S. M. Reddy. Distributed fault-tolerance for large multiprocessor systems. In *Proceedings of the 7th annual symposium on Computer Architecture*, pages 23–30. ACM, 1980.

[86] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[87] B. W. Lampson. Atomic transactions. In *Distributed Systems - Architecture and Implementation, An Advanced Course*, pages 246–265, London, UK, UK, 1981. Springer-Verlag.

[88] J.-C. Laprie. Dependable computing: Concepts, limits, challenges. In *Proceedings of the Twenty-Fifth International Conference on Fault-tolerant Computing*, FTCS'95, pages 42–54, Washington, DC, USA, 1995. IEEE Computer Society.

[89] P. A. Lee and T. Anderson. *Fault tolerance: principles and practice*, volume 3. Springer Science & Business Media, 2012.

[90] P. Lemarinier, A. Bouteiller, T. Herault, G. Krawezik, and F. Cappello. Improved message logging versus improved coordinated checkpointing for fault tolerant mpi. In *Cluster Computing, 2004 IEEE International Conference on*, pages 115–124, Sept 2004.

[91] R. Luling, B. Monien, and F. Ramme. Load balancing in large networks: a comparative study. In *Parallel and Distributed Processing, 1991. Proceedings of the Third IEEE Symposium on*, pages 686–689, Dec 1991.

[92] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[93] C. Men, Z. Xu, and X. Li. An efficient checkpointing and rollback recovery scheme for cluster-based multi-channel ad hoc wireless networks. In *Parallel and Distributed Processing with Applications, 2008. ISPA'08. International Symposium on*, pages 371–378. IEEE, 2008.

[94] M. M. Michael. Aba prevention using single-word instruction. Technical report, Thomas J. Watson Research Center, P.O. Box 218 Yorktown Heights, NY 10598, Jan 2004.

[95] V. Mikolasek and H. Kopetz. Roll-forward recovery with state estimation. In *Proc. 14th IEEE International Symposium on Object-oriented Real-time distributed Computing*, pages 179–186, Newport Beach, CA, USA, Mar. 2011.

[96] N. Mohammed, H. Otrok, L. Wang, M. Debbabi, and P. Bhattacharya. Mechanism design-based secure leader election model for intrusion detection in manet. *Dependable and Secure Computing, IEEE Transactions on*, 8(1):89–103, 2011.

[97] M. Moir and J. H. Anderson. Wait-free algorithms for fast, long-lived renaming. *Science of Computer Programming*, 25(1):1 – 39, 1995.

[98] V. Nelson and B. Carroll. *Tutorial: fault-tolerant computing*. IEEE Computer Society Press, 1987.

[99] V. Nguyen and R. Kirner. Demand-based scheduling priorities for performance optimisation of stream programs on parallel platforms. In *Algorithms and Architectures for Parallel Processing*, volume 8285 of *Lecture Notes in Computer Science*, pages 357–369. Springer International Publishing, 2013.

[100] V. T. N. Nguyen. *An Efficient Execution Model for Reactive Stream Programs*. PhD thesis, University of Hertfordshire, 2014.

[101] V. T. N. Nguyen, R. Kirner, and F. Penczek. A multi-level monitoring framework for stream-based coordination programs. In *Proc. 12th International Conference on Algorithms and Architectures for Parallel Processing*, LNCS, Fukuoka, Japan, Sep. 2012. Springer.

[102] U. of Hertfordshire. Compiler technology and computer architecture group.

[103] J. Park, M. Smelyanskiy, K. Vaidyanathan, A. Heinecke, D. D. Kalamkar, X. Liu, M. M. A. Patwary, Y. Lu, and P. Dubey. Efficient shared-memory implementation of high-performance conjugate gradient benchmark and its application to unstructured matrices. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 945–955, Piscataway, NJ, USA, 2014. IEEE Press.

[104] T. Park and H. Yeom. An asynchronous recovery scheme based on optimistic message logging for mobile computing systems. In *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on*, pages 436–443, 2000.

[105] T. M. Parks. *Bounded scheduling of process networks.* PhD thesis, University of California, 1995.

[106] F. Penczek, J. Julku, H. Cai, P. Hölzenspies, C. Grelck, S.-B. Scholz, and A. Shafarenko. S-Net language report (Version 2.0). Technical Report No. 499, University of Hertfordshire, Hatfield, United Kingdom, Apr. 2010.

[107] G. L. Peterson and M. J. Fischer. Economical solutions for the critical section problem in a distributed system. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 91–97. ACM, 1977.

[108] R. Prakash and M. Singhal. Low-cost checkpointing and failure recovery in mobile computing systems. *IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS*, 7:1035–1048, 1994.

[109] D. Prokesch. A light-weight parallel execution layer for shared-memory stream processing. Master's thesis, Technische Universität Wien, Vienna, Austria, Feb. 2010.

[110] A. Pullini, F. Angiolini, D. Bertozzi, and L. Benini. Fault tolerance overhead in network-on-chip flow control schemes. In *Integrated Circuits and Systems Design, 18th Symposium on*, pages 224–229. IEEE, 2005.

[111] B. Randell. System structure for software fault tolerance. *SIGPLAN Not.*, 10(6):437–449, Apr. 1975.

[112] S. Rao, L. Alvisi, and H. Vin. The cost of recovery in message logging protocols. *Knowledge and Data Engineering, IEEE Transactions on*, 12(2):160–173, Mar 2000.

[113] S. Rao, L. Alvisi, and H. M. Vin. Egida: An extensible toolkit for low-overhead fault-tolerance. In *Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on*, pages 48–55. IEEE, 1999.

[114] M. Raynal. *Concurrent Programming: Algorithms, Principles, and Foundations.* Springer, Heidelberg, 2013.

[115] D. A. Reed, C. L. Mendes, et al. Reliability challenges in large systems. *Future Generation Computer Systems*, 22(3):293–302, 2006.

[116] I. Ring Nielsen. Constraint and application driven framework for tailoring embedded real-time systems, 2012.

[117] B. Schroeder, G. Gibson, et al. A large-scale study of failures in high-performance computing systems. *Dependable and Secure Computing, IEEE Transactions on*, 7(4):337–350, 2010.

[118] H. Sekhar Paul, A. Gupta, and A. Sharma. Finding a suitable checkpoint and recovery protocol for a distributed application. *J. Parallel Distrib. Comput.*, 66(5):732–749, may 2006.

[119] A. Shafarenko. Nondeterministic coordination using s-net. In W. Gentzsch, L. Grandinetti, and G. Joubert, editors, *High Speed and Large Scale Scientific Computing*, volume 18 of *Advances in Parallel Computing*, pages 74–96. IOS Press, 2009.

[120] M. Shirali, A. Toroghi, and M. Vojdani. Leader election algorithms: History and novel schemes. In *Convergence and Hybrid Information Technology, 2008. ICCIT '08. Third International Conference on*, volume 1, pages 1001–1006, Nov 2008.

[121] G. S. Sohi, M. Franklin, and K. K. Saluja. A study of time-redundant fault tolerance techniques for high-performance pipelined computers. In *Fault-Tolerant Computing, 1989. FTCS-19. Digest of Papers., Nineteenth International Symposium on*, pages 436–443. IEEE, 1989.

[122] R. E. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3:204–226, 1985.

[123] Y. Tamir and C. H. Squin. Error recovery in multicomputers using global checkpoints. In *In 1984 International Conference on Parallel Processing*, pages 32–41, 1984.

[124] A. S. Tanenbaum and M. v. Steen. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[125] W. Torell and V. Avelar. Mean time between failure: Explanation and standards. *White Paper*, 78, 2004.

[126] M. Treaster. A survey of fault-tolerance and fault-recovery techniques in parallel systems. *CoRR*, abs/cs/0501002, 2005.

[127] J. Tromp and P. Vitányi. Randomized two-process wait-free test-and-set. *Distributed Computing*, 15(3):127–135, 2002.

[128] P. Tsigas and Y. Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '01, pages 134–143, New York, NY, USA, 2001. ACM.

[129] P. Veríssimo and L. Rodrigues. *Distributed Systems for System Architects*. Advances in Distributed Computing and Middleware. Springer US, 2012.

[130] D. Yadav, C. Lamba, and S. Shukla. A new approach of leader election in distributed system. In *Software Engineering (CONSEG), 2012 CSI Sixth International Conference on*, pages 1–7, Sept 2012.

Appendix A

Appendix 1

A.1 Experiments Data

This appendix contains the results of the experiments described in Section 8.

Leader election experiments	4 cores	8 cores	16 cores	32 cores	48 cores
1	0.07	0.27	0.26	0.81	1.2
2	0.1	0.21	0.34	0.92	1.4
3	0.09	0.17	0.47	0.78	1.24
4	0.08	0.23	0.49	0.84	1.34
5	0.07	0.15	0.44	0.95	1.56
6	0.11	0.25	0.2	0.82	1.62
7	0.13	0.31	0.37	0.77	1.22
8	0.09	0.19	0.38	0.79	1.33
9	0.07	0.13	0.27	0.86	1.29
10	0.16	0.23	0.5	0.91	1.43
11	0.13	0.19	0.25	0.73	1.59
12	0.16	0.21	0.33	0.83	1.62
13	0.13	0.24	0.34	0.97	1.41
14	0.15	0.25	0.43	0.91	1.51
15	0.15	0.29	0.28	0.94	1.58
16	0.09	0.21	0.43	0.86	1.43
17	0.15	0.25	0.19	0.81	1.52
18	0.1	0.22	0.4	0.74	1.44
19	0.15	0.19	0.32	0.92	1.71
20	0.14	0.14	0.23	1.01	1.42
21	0.15	0.25	0.3	0.87	1.53
22	0.15	0.15	0.23	0.77	1.48
23	0.16	0.15	0.39	0.87	1.36
24	0.14	0.13	0.27	0.91	1.61
25	0.13	0.12	0.51	0.84	1.41
26	0.1	0.14	0.48	0.85	1.41
27	0.16	0.29	0.25	0.77	1.14
28	0.16	0.28	0.23	0.9	1.69
29	0.17	0.26	0.34	0.82	1.37
30	0.16	0.2	0.47	0.78	1.54
31	0.08	0.28	0.43	0.79	1.44
32	0.14	0.12	0.46	0.92	1.39
33	0.11	0.28	0.33	0.84	1.47
34	0.16	0.27	0.29	0.99	1.37
35	0.17	0.31	0.23	0.97	1.52
36	0.15	0.22	0.46	0.87	1.58
37	0.14	0.15	0.54	0.88	1.78
38	0.08	0.15	0.51	0.93	1.39
39	0.18	0.12	0.49	0.9	1.23
40	0.1	0.3	0.52	0.76	1.49
41	0.14	0.33	0.26	0.92	1.53
42	0.11	0.15	0.42	0.8	1.61
43	0.13	0.13	0.54	1.01	1.23
44	0.16	0.22	0.36	0.96	1.56
45	0.14	0.37	0.26	0.81	1.69
46	0.16	0.13	0.55	0.79	1.39
47	0.09	0.12	0.4	0.84	1.46
48	0.13	0.28	0.52	0.98	1.43
49	0.14	0.25	0.36	0.91	1.38
50	0.11	0.27	0.22	0.89	1.4
AVG	0.1284	0.215	0.3708	0.8662	1.4548

Table A.1. Execution time of leader election experiments

Reading experiments	4 cores	8 cores	16 cores	32 cores	48 cores
1	17	42	91	184	279
2	18	40	90	186	282
3	16	41	87	180	276
4	17	39	91	178	274
5	18	42	85	180	284
6	16	39	87	175	283
7	17	40	86	179	281
8	16	39	87	182	285
9	16	37	88	183	280
10	16	39	85	181	276
11	17	41	87	174	278
12	16	38	88	184	277
13	17	39	85	180	279
14	16	42	87	186	283
15	16	37	88	185	270
16	17	39	86	173	281
17	16	38	87	181	280
18	16	39	88	179	285
19	17	38	85	183	273
20	16	39	87	176	279
21	18	42	89	178	273
22	16	38	90	180	275
23	17	39	87	181	276
24	18	41	88	185	280
25	16	39	87	183	273
26	16	39	90	182	284
27	16	38	85	186	276
28	18	39	87	185	274
29	16	39	85	182	275
30	16	37	88	179	273
31	17	38	87	183	274
32	18	42	86	181	275
33	16	39	87	176	271
34	16	37	89	178	270
35	16	38	87	177	276
36	18	39	88	180	271
37	16	42	92	179	270
38	16	38	87	178	276
39	17	37	91	179	273
40	16	39	87	175	270
41	16	39	87	174	278
42	17	37	85	183	276
43	16	39	86	186	270
44	18	37	87	181	276
45	16	39	85	176	275
46	16	41	87	179	270
47	18	39	86	187	281
48	17	42	88	183	270
49	18	38	87	181	284
50	16	39	90	180	281
AVG	16.63265306	39.10204082	87.32653061	180.4489796	276.5714286

Table A.2. The amount of reading by nodes to the shared memory in the leader election experiments

Log-based rollback-recovery experiments	4 cores	8 cores	16 cores	32 cores	48 cores	Average
Time without fault tolerance	124.57	260.7	630.85	1235.72	1582.93	766.954
Time with fault tolerance and without failures	135.04	282.94	688.26	1345.69	1722.64	834.914
Time with fault tolerance and one failure	139.78	290.67	708.89	1393.32	1794.36	865.404
Time with fault tolerance and several failures	149.4	319.93	790.47	1573.71	2046.82	976.066
Percentage without failure	8.4049129	8.530878404	9.100420068	8.899265206	8.826037791	8.752302874
Percentage with one failure	12.21000241	11.49597238	12.37061108	12.75369825	13.35687617	12.43743206
Percentage with failures	19.93256803	22.71960107	25.30236982	27.35166543	29.30578105	24.92239708

Table A.3. The log-based rollback-recovery experiments results

Percentage of failure can appear each time a stream is read or written	4 cores	8 cores	16 cores	32 cores	48 cores
0.8 in 1000	140.2	288	700	1430	1900
0.9 in 1000	141.3	301	750	1500	1950
1 in 1000	149.4	319.93	790.47	1573.71	2046.82
2 in 1000	155	331	805	1630	2140
3 in 1000	161	350	828	1680	2222
4 in 1000	170	379	880	1800	2431

Table A.4. Results of changing the percentage of failure into the log-based rollback-recovery experiments

Registers used by restore mechanism	Percentage of execution time overhead
START	5.5
READ	3.512
WRITE data was not written to the stream	9.49
WRITE data was written to the stream	6.15
BLOCK IsTaskReceived is true	0.493
BLOCK IsTaskReceived is false	3.45
END IsTaskReceived is true	1.623
END IsTaskReceived is false	2.65

Table A.5. Percentage of execution time overhead caused by the registers used by the restore mechanism proposed