# CAST: Design of the Algorithmic Framework for Actor-Based Cloud Simulation

BY

AMEDEO BARAGIOLA
B.S., Politecnico di Milano, Milan, Italy, 2018

THESIS

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2021

Chicago, Illinois

Defense Committee:

Mark Grechanik, Chair and Advisor
Balajee Vamanan
Elisabetta Di Nitto, Politecnico di Milano

# ACKNOWLEDGMENTS

First, I want to thank my advisor Dr. Mark Grechanik for his support and suggestions throughout the duration of the work we conducted together and for giving me the opportunity to work as a Graduate Research Assistant under his guidance. His advice and inputs have helped me greatly in writing my thesis. This would not have been possible without his guidance and support. I also want to thank Professor Elisabetta Di Nitto from Politecnico di Milano and Professor B. Vamanan for serving on my committee.

AB

# TABLE OF CONTENTS

# TABLE OF CONTENTS (continued)

# LIST OF TABLES

# LIST OF FIGURES

# LIST OF ABBREVIATIONS

| | |
|---|---|
| SimEng | Simulation Engineer |
| CAST | Cloud-based Applications Simulator Testbed |
| CASTDSL | CAST Declarative Specification Language |
| VM | Virtual Machine |

# SUMMARY

Nowadays, it is increasingly important for researchers to understand and predict the behaviour of large scale cloud datacenters and to analyze novel distributed algorithms running on top of them. Studying large scale cloud datacenter architectures helps in solving production issues in current cloud environments and in evaluating the performance of newly proposed algorithms. However, building large scale cloud datacenters is very expensive and working on existing real-world infrastructure does not provide the flexibility necessary to carry out experiments and test new algorithms. Therefore large scale simulations are fundamental to evaluating distributed algorithms and understanding their behaviour. In this thesis we introduce CAST, a Cloud-based Applications Simulator Testbed, which makes large-scale simulations possible in a simple and intuitive way. In particular, we start from the Actor-Based architecture originally proposed for CAST and we expand it by integrating high-level functionalities such as the Global Clock, Consistency and Replication algorithms, Election Algorithms and Partitioning and Failure modeling. We provide an algorithmic framework for integrating such functionalities, fundamental to all cloud simulators, with an Actor-Based system while building upon CASTDSL, an easy to use declarative language for simulation engineers, to provide intuitive access to the introduced high-level features even with limited programming experience or knowledge of the underlying architecture.

# CHAPTER 1

# INTRODUCTION

## 1.1 Overall Context and Research Problem

With the projection of the worldwide cloud computing to grow to a half trillion USD by 2023, it is very important for systems researchers to experiment with novel algorithms and systems optimizations in complex cloud datacenters in order to deeply understand and predict their behavior for the advancement of science and for societal progress [1]. Evaluating new algorithms and techniques in large-scale cloud organization settings is vitally important, since new unforeseen phenomena can be uncovered. Researchers can convince industry to adopt their ideas only if their experimental setup captures scale, heterogeneity, and other such characteristics of today's large datacenters. Unfortunately, building large cloud datacenters is very expensive and even with existing ones it is very difficult to carry out experiments to determine the effectiveness and efficiency of various algorithms and techniques that involve millions of cloud users and servers to evaluate research ideas. Operating large cloud datacenters can be highly sensitive to a number of interdisciplinary aspects such as the choice of application frameworks and their schedulers, revenue management models, memory consistency models, and networking and storage technologies. In addition, a cloud organization is comprised of multiple regions, which include several datacenters each of which may contain more than $10,000$ racks with 100 computer servers on each rack. Even a moderate-sized cloud organization hosts millions of servers spread across multiple datacenters in different geographic regions with annual operating expenditures of over $18 million each [2; 3].

Because the expense and the scale of cloud datacenters make it very difficult for researchers to create and experiment with distributed algorithms for various aspects of cloud computing, the researchers often resort to small-scale experiments or formal proofs of correctness of these algorithms. However, without experimenting with real-world system models, the proofs alone do not provide enough confidence that distributed algorithms will perform as desired in a large-scale cloud organization. Since cloud computing is a key CISE topic, its research and education need evaluations of various networking and software optimizations on real hardware in small custom-built clouds, and analysis of the applications' behaviours in small clusters that run framework like Hadoop's map/reduce or Spark [4; 5; 6; 7]. In all, the inability to conduct large-scale comprehensive experiments with distributed algorithms stunts the growth of research and education in cloud computing.

## 1.2 Our Contributions

This thesis work builds upon the Actor-Based architecture originally proposed for CAST, a Cloud-based Applications Simulator Testbed, and expands it by integrating high-level functionalities fundamental to all cloud simulators.

In particular, we identify the following contributions:

1. We briefly investigate the state-of-the-art of cloud simulators that we can use as the underlying architecture for this work. For each simulator we focus on:
    - Its ability to run large scale simulations.

- Its extensibility for researchers who wish to add additional features to the simulator.
- Its ease of use for SimEngs unaware of the simulator architecture.

2. We analyze the availability of implementations of algorithms related to Clocks, Consistency, Election algorithms and Partitioning and Failure modeling in existent simulators.
3. When we find existing implementations in the literature we focus on the constraints that such implementations impose on users like limited features or restrictions on the scenarios in which the implementations are applicable. (e.g. only in MapReduce algorithms See section 7.2.3).
4. We identify the steps to implement Global clocks, Consistency and Replication algorithms, Election Algorithms and Partitioning and Failure modeling in an Actor-based architecture including defining data structures in every node, implementing a protocol to update these data structures, defining message types exchanged among actors, providing out-of-the-box algorithms and APIs to SimEngs and, when necessary, comparing such algorithms to identify drawbacks and advantages of each.

# CHAPTER 2

# MOTIVATION

Currently, research in cloud computing in general relies heavily on simulations to evaluate distributed algorithms that govern various aspects of the cloud application deployment. The closer the simulated cloud represents the actual cloud organization and its behavior, the more convincing the results of the simulation become with experimental distributed algorithms [8]. The ability to avoid a significant expense of running large-scale experiments in the actual cloud organization while exploring and understanding the behavior of customized cloud architecture has been a Holy Grail in the field of cloud computing. A few dozen cloud computing simulators have been build over the past decade [9], however, there are no records of detailed simulations that allow researchers and educators to experiment with millions of servers in multiple datacenters and various networking devices and finely modeled applications [10; 11]. Moreover, using multithreading with synchronization mechanisms to protect concurrent accesses to shared objects in simulation programs resulted in bottlenecks and inability to scale to very large simulations, which unfortunately, are extremely demanding in resources including but not limited to processor's speed and the memory size in addition to data transfer rates among these resources. More importantly, various simulators (e.g., CloudSim [12]) offer rigid design frameworks that are difficult to learn and use, and the created simulations cannot be easily transferred among different simulation frameworks, making their reproducibility, reuse and scaling very difficult.

Distributed and cloud computing research often requires realistic workload models resulting from running millions of applications which read and write distributed objects. While specific requirements for these applications differ, they can be represented as collections of jobs that in turn consist of various tasks that perform read and write operations. For example, a map/reduce application consists of mapping and reducing jobs that perform various tasks on key/value pairs and these tasks read input pairs and write output ones. Depending on specific properties of these applications, which can be modeled as probabilistic distributions of read and write operations, distributed algorithms for load balancing and resource allocation may perform differently depending not only on the underlying network infrastructure topology, but also on what hardware is used to perform these operations and host distributed objects on which these operations are performed. Unfortunately, no existing cloud simulators allow researchers to plug different characteristics of the underlying hardware with respect to their effect on the read and write operations, let alone to plug their simulators into the actual cloud to collect this information from the cloud dynamically in a self-correcting optimizing feedback loop. Doing so would enable systems researchers to quickly test their ideas and improve them based on the highly relevant feedback from such a cloud simulator.

To summarize, problems with all existing cloud simulators can be described across three interlocking dimensions: performance, efficiency and code engineering [13; 14]. To improve performance of cloud simulators, they should be able to avoid creating internal bottlenecks (e.g., using synchronization mechanisms) while scaling by running on multiple computing nodes without having to redesign the code. In fact, simulations and cloud computing are tightly connected:

simulators are designed to run in the cloud to gain performance by exploiting its parallelism and elasticity whereas cloud simulators simulate cloud computing [15; 16; 17; 18; 19; 20; 21; 22; 23; 24; 25; 26; 27; 28; 29; 30]. Efficiency can be achieved by aggressively reducing the sizes of all internal objects created and used as part of the simulation, which is difficult to achieve when objects are required to inherit their functionalities from fat classes located in the underlying simulation framework. Finally, the biggest problem is to create maintainable and reusable simulations quickly and without having to learn special programming languages and sprawling fat object-oriented frameworks. Creating new cloud simulators is often a difficult code engineering problem which requires significant human resources and time to go through the entire software engineering lifecycle to create working simulators.

We will now introduce what a distributed system is and start building upon the Actor-Based architecture originally proposed for CAST, a Cloud-based Applications Simulator Testbed, by describing and implementing, along with the respective messages and APIs, key features like the Global Clock, Consistency and Replication algorithms, Election Algorithms and Partitioning and Failure modeling while building upon CASTDSL, an easy to use declarative language for simulation engineers, to provide intuitive access to the introduced high-level features even with limited programming experience or knowledge of the underlying architecture.

# CHAPTER 3

# THE GLOBAL CLOCK

”A distributed system is a collection of independent computers that appears to its users as a single coherent system.” [31] In CAST, we model distributed systems as a graph where each node is a computing entity that can be represented as an actor and that performs read and write operations on some shared resource stored on one or multiple nodes. A shared resource is some data item that is read and/or written by more than one node of the distributed system. Every node exists in its own memory space[1] and accesses the memory space of the other nodes via messages sent over a network channel.

*Example* 3.0.1*.* For example a Database server split across multiple nodes, or servers, is a distributed system. The nodes on which the system runs are connected by network links which represent the connections in the graph.

**Definition 3.0.1** (Process)**.** A process is a single instance of a program which emits a sequence of read and write operations. (messages)

*Remark. For the sake for simplicity we assume that at any given time on each of the nodes at most one process is running. This assumption is not realistic, but it is useful to simplify the*

---

[1] **Memory space**: The memory that a node can can directly read and write using the assigned addresses of the memory cells.

*definitions given and the concepts introduced in this chapter. When we state that "a node is executing some operation", we mean that the single process running on that node executes that operation. In other words a node is a process, there is a one-to-one mapping between the two. Furthermore, we observe how this assumption doesn't impose any fictitious or unrealistic constraint as considering more processes on a single node or multiple nodes with a single process each is equivalent for the purposes of defining the clock.*

In traditional single-threaded monolithic applications, where all the read and write operations are performed in the same memory space, operations are executed sequentially and resources are manipulated one at a time. In distributed systems messages are exchanged concurrently and the same resources are manipulated asynchronously by multiple nodes. When multiple messages access the same resource at the same time the order of such messages (and consequently of the operations) directly influences the output of the system.

**Definition 3.0.2** (Operation)**.** An Operation is a 3-tuple $\langle x, y, z \rangle$ (written an xyz) s.t. $x \in \{W, R\}; 1 \leq y \leq n; z \in R$, where z is a resource, y is the node identifier, $n$ is the total number of nodes in the distributed system, $R$ is the set of shared resources, $W$ represents an operation which changes the resource it operates on (a write operation) and $R$ represents operations which don't change the resources (read operations) [31].

The 3-tuple $\langle x, y, z \rangle$ is read as: $x$ is executed on node $y$ and on the subset of resources $z$.

*Example* 3.0.2. Let us consider the following order of operations on the same resource x, x is a "char" type variable: W1x=a, W2x=b, R3x, R4x. The semantics of the operations above is as follows: Operation W1x writes the value 'a' into x ($x = a$) and is executed on node 1; W2x writes the value 'b' into x ($x = b$) and is executed on node 2; R3x reads x on node 3; R4x reads x on node 4. Different orders of those operations, or equivalent serial schedules, yield different outputs for R3x and R4x. For instance if we consider the following serial schedule W1x, W2x, R3x, R4x, the output of R3x is 'b' and R4x is 'b'. If the schedule is instead W1x, R3x, W2x, R4x, then node 3 sees the value 'a' for x, while node 4 sees the value 'b'. Seeing a value means that when node 3 accesses the resource 'x' it is returned the value 'a' since the latest write is W1x; while when node 4 access the resource x, the value 'b' is now returned since the latest write is W2x.

**Definition 3.0.3** (Race condition)**.** A race condition is a situation where two or more operations are executed concurrently, whereas they should have been executed in a certain order, producing unexpected results.

In a distributed system if we access the same resource at the same time from multiple nodes, we may encounter a race condition since the order in which the operations are executed becomes non-deterministic. We want to avoid mistakenly returning different values for a resource depending on the order in which operations are performed, like we have shown above, so we need to introduce the concept of causality (causal precedence relation) between events and establish at minimum a partial ordering of operations.

**Definition 3.0.4** (Event)**.** An Event is:

- The execution of an operation by the current node: a read or a write operation.
- Sending a message to another node.

- Receiving a message from another node.

**Definition 3.0.5** (Causal precedence relation)**.** Relation →, known as the "happened before" or "causal precedence" relation, is a relation defined on the set of events (or operations) in a system satisfying the following three conditions [32]:

1. "If a and b are events on the same process and a comes before b, then a → b" [32].
2. "If a is an outgoing message sent from one node to another and b is the receipt of such message by the other node, then a → b" [32].
3. "If a → b and b → c, then a → c" [32].

In traditional systems causality is tracked with physical time: if a → b, then the time at which $b$ is recorded is greater than the time at which $a$ is recorded. By physical time we mean the time expressed as the number of milliseconds passed since 1970-01-01 00:00:00 UTC, this is also known as "system time". A process gets the current time by issuing a syscall to the kernel through the operating system.

In distributed systems it's not possible to have a global time that is the same across all nodes because each node keeps track of physical time separately and the syscall above yields a different result on different nodes; we can synchronize the physical time on each node that makes up the system within a certain precision $\delta$, for example by relying on protocols such as the Network Time Protocol (NTP) that sync the system clock from a remote time server and take into account the network latency, however, due to the intrinsic nature of clocks (an imperfect, real-world oscillator) even if we try to synchronize the physical time on each node that makes up the system, it may drift. Therefore, we can't rely on physical time to establish causality between events. [33]

*Example* 3.0.3. Consider a distributed system composed of two nodes. On the first node (A) the system time is 2020-11-26T00:26:21+00:00. On the second node (B) it is 2020-11-26T00:26:08+00:00. A write operation ($w_A$) is executed on node A, and a read operation ($r_B$) is executed on node B, furthermore $w_A \rightarrow r_B$ holds.

Let us say this distributed system uses the physical time to establish causality between events. Node A executes $w_A$ and sends it over to node B along with the timestamp at which it was executed 2020-11-26T00:26:21+00:00. At the same time node B executes $r_B$ with timestamp 2020-11-26T00:26:09+00:00 and then receives $w_A$ from node A.

Node B can't reliably order the events, it would think that $w_A$ was executed after $r_B$ as it has a higher timestamp, but in reality $w_A \rightarrow r_B$.

Since we can't use the concept of physical time, we need to introduce a different type of time: the logical time.

**Definition 3.0.6** (Logical clock)**.** The Logical clock is a function from the set of events to the time domain, represented as CLK(e), such that the following holds true [33]: $for\ each\ e_i, e_j : e_i \rightarrow e_j,\ then\ CLK(e_i) < CLK(e_j)$. A logical clock for which this definition holds is known as a consistent clock.

## 3.1 Implementation framework for Logical Clocks

In this section we will introduce the steps necessary to implement logical clocks in distributed systems, including CAST. We will later describe how these steps are mapped to the CAST code for each type of logical clock we consider.

Implementing Logical clocks in distributed systems, including CAST, means solving the following two issues [33]:

1. Implementing data structures in every node to represent the logical clock of that node.
2. Implementing a protocol to update these data structures in a way such that the clock is consistent. (See definition 3.0.6)

More in detail, for what concerns the data structures, we need to [33]:

- Allow the node to keep track of its own progress. This means keeping track of the events it has processed so far. This is known as implementing the local logical clock.
- Represent the node view of the global logical time. This is known as implementing the global logical clock.

Also, we need to define a protocol that [33]:

- Defines how the local logical clock is updated when the node processes a new event (See definition 3.0.4)
- Defines how a node updates its global logical clock in response to events.

## 3.2 Lamport scalar clock

The Lamport scalar clock was introduced by Lamport in 1978 as an attempt to provide a way to fully order events in a distributed system. [34] We will now show how to enable the Lamport scalar clock for a certain node, then we will define the Lamport clock in terms of data structures and protocol used to update those data structures and show their implementation in CAST.

### 3.2.1 Enabling the Lamport scalar clock

When the SimEng wants to use this type of clock for a certain node, he needs to specify a node-level property in CASTDSL as follows:

```
my_server_0:
        ...
        capabilities:
            host:
                    clock_type:
                        type: ClockAlgorithm
                        value: ScalarLamport
        ...
```

Listing 3.1: Sample SimpleNode construct. The SimEng associated LC to nodes in the system. The "ScalarLamport" keyword enables the data structures and clock protocol below.

### 3.2.2 Data structures

The Lamport scalar clock uses a single integer variable which is used to track both the local logical clock and the global logical clock. [33] In CAST this is implemented by a single variable $clk_{lamport}$ inside each node state:

| Key | Value | Type |
|---|---|---|
| _role | "host" | 0 |
| ... | ... | ... |
| $\_clk_{lamport}$ | 0 | 0 |

TABLE I: State of node with Lamport clock.

The table above represents the state of a node in the graph which in CAST is modeled as a Map: $Key \longrightarrow \langle Value, Type \rangle$ where Key, Value and Type are column names.[35] The Lamport Clock value is stored at the Key $\_clk_{lamport}$ and has an initial value of 0 (initialization value) along with a type of 0 indicating that this entry is a "SimpleResource". This representation of the state, including the definition of SimpleResource(s), is referenced from *M. Pelosi, "CAST: Cloud-based applications simulator testbed."* where it is described in more detail.

### 3.2.3 The Format of Messages

When we employ the Lamport clock in a distributed system we can formalize the messages exchanges in such system as tuples with the message content and the clock value (an integer) at the time at which the message was emitted as tuple elements: $\langle content, clk_{msg} \rangle$

### 3.2.4 Clock update protocol

The Lamport clock specifies that when the node processes a new event the local logical clock must be increased by 1: $clk_{lamport} = clk_{lamport} + 1$. [33] In CAST we obtain this by incrementing the state variable representing the clock: state.clk$_lamport = state.clk_lamport+$1; Additionally, when a new event is processed the node's view of the global logical time is updated as follows:

1. $clk_{lamport} = max(clk_{lamport}, clk_{msg})$: The clock is update with the maximum value between the clock value received along with the message and the current value of the clock.
2. $clk_{lamport} = clk_{lamport} + 1$
3. Ready to process more events, including sending or receiving messages.

The CAST implementation follows:

```
override def receiveCommand: Receive = {
case event =>
state.clk_lamport = max(state.clk_lamport, event.c_msg)
state.clk_lamport = state.clk_lamport + 1;
processEvent(event);
}
```

The receiveCommand function (present in each Actor) is invoked by the Akka framework upon receipt of a new event, the state variable representing the clock is updated according to the clock specification (described above) and then the event is handled by the processEvent function.

### 3.2.5 Shortcomings of the Lamport clock

The Lamport clock was introduced as a first attempt to fully order events in a distributed system, but falls short of this goal. By using a single integer variable to keep track of the local logical clock and using the same variable to represent the node view of the global logical time and by implementing the protocol described above, such clock shows two main shortcomings:

- In regards to the aim of establishing a total order of events, two or more events in two different nodes can have the same clock value associated to them (therefore they are not totally ordered).
- It does not satisfy the clock strong consistency property defined below.

**Definition 3.2.1** (Clock strong consistency)**.** A Logical clock (see Definition 3.0.6) is said to be strongly consistent if and only if the following holds true [33]: $for\ each\ e_i, e_j : e_i \rightarrow e_j \Longleftrightarrow CLK(e_i) < CLK(e_j)$.

A clock that satisfies the clock strong consistency property lets us derive the causality of events simply by looking at their respective $CLK(e)$ clock values. In other words, take two events $e_1, e_2$ if the comparison between $CLK(e_1)$ and $CLK(e_2)$ evaluates to: $CLK(e_1) < CLK(e_2)$, then we can assert that $e_1 \rightarrow$e$_2$: $e_1$ happened before $e_2$.

To better illustrate the shortcomings on the Lamport clock, we consider the following scenario, we illustrate how this is consistent with the clock definition, and show the issues that arise.

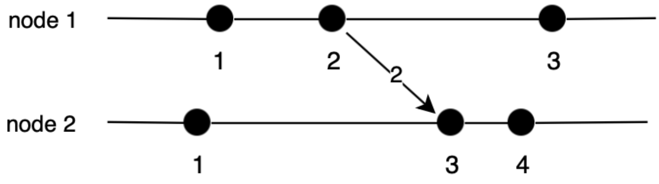

Figure 1: Lamport clock execution

The following is the list of instructions which lead to the scenario above:

1. Both node1 and node2 start with $clk_{lamport} = 0$.
2. node1 processes a new event and increments $clk_{lamport}$ by 1.
3. node2 processes a new event and increments $clk_{lamport}$ by 1.
4. node1 wants to send a message to node2: it increments the $clk_{lamport}$ again and then sends the message.
5. node2 receives a message from node1: it computes the max value between its current clock value (1) and the clock value received with the message from node1 (2) and sets its clock value to 2.
6. node2 increments its clock value by 1 and then processes the event it has received (with clock value 3).
7. node1 processes a new event and increments $clk_{lamport}$ by 1. (now 3)
8. node2 processes a new event and increments $clk_{lamport}$ by 1. (now 4)

From this example we can highlight the two shortcomings indicated above:

- Two events, respectively the third and second event to node1 and node2 have the same clock value associated to them.
- The third event on node1 has a lower associated clock value than the third event on node2, but the "happens before" relation does not hold. These events are independent and there is no causal precedence among them, however, by using the Lamport clock we impose a fictitious order on them.

## 3.3 Tie resolution in Lamport Clocks

As mentioned in section 3.2.5 one of the shortcomings on the Lamport Clock is that two or more casually related events representing a replica and its master have the same clock value. This is a problem because it is impossible to create a total order of such events using the Lamport clock. In practical implementations the Lamport Clock is combined with a tie resolution mechanism to avoid such scenarios where two events have the same clock value. [31]

In CAST a textbook mechanism is employed: when such conflicts arise, we order events with the same timestamp by node id, such that if $CLK(e_1) = CLK(e_2) \wedge nodeid(e_1) < nodeid(e_2)$ we can assert that: $e_1 \rightarrow \mathrm{e}_2$. And, if $CLK(e_1) = CLK(e_2) \wedge nodeid(e_1) > nodeid(e_2)$ we can assert that: $e_2 \rightarrow \mathrm{e}_1$, where *nodeid* is a function that returns the id of the node the event is executed on.

## 3.4 Vector clock

The first vector clocks were introduced indipendently by Fidge, Mattern and Schmuck and aim at resolving the issues presented by scalar clocks such as the Lamport clock.

[33] As we did for the Lamport clock, we will now show how to enable the Vector clock for a certain node, then we will define it in terms of data structures and protocol used to update those data structures and show their implementation in CAST.

### 3.4.1 Enabling the Vector clock

When the SimEng wants to use this type of clock for a certain node, he needs to specify a node-level property in CASTDSL as follows:

Listing 3.2: Sample SimpleNode construct, focus on clock property.

```
my_server_0:
        ...
        capabilities:
            host:
                    clock_type:
                        type: ClockAlgorithm
                        value: VectorClock
        ...
```

The ”VectorClock” keyword enables the data structures and clock protocol below.

### 3.4.2 Data structures

The Vector clock uses a vector of non-negative integers to keep track of time. Each node $i$ has a vector $clk_{vect_i}[1..n]$ where $n$ is the total number of nodes in the distributed system and $i$ is the index of the node. $clk_{vect_i}[i]$ is the local logical clock of the node $i$ and represents the progress of the current node. $clk_{vect_i}[j]$ represents the knowledge that node $i$ has of the progress of node $j$.

The full vector $clk_{vect_i}[1..n]$ corresponds to the node view of the global logical time. [33] In CAST this is implemented by storing the clock vector $clk_{vect_i}[1..n]$ inside each node state:

| Key | Value | Type |
|---|---|---|
| _role | "host" | 0 |
| ... | ... | ... |
| $\_clk_{vect_i}$ | $[0..0]_n$ | 0 |

TABLE II: State of node i with Vector clock.

The table above represents the state of the node in the graph which in CAST is modeled as a Map: $Key \longrightarrow \langle Value, Type \rangle$ where Key, Value and Type are column names.[35] The Vector Clock value is stored at the Key $\_clk_{vect_i}$ and is initialized to a vector of zeros of length $n$ where $n$ is the number of nodes in the group. Its type is set to 0 indicating that this entry is a "SimpleResource". This representation of the state, including the definition of SimpleResource(s), is referenced from *M. Pelosi, "CAST: Cloud-based applications simulator testbed."* where it is described in more detail.

### 3.4.3 The Format of Messages

When we employ the Vector clock in a distributed system we can formalize the messages exchanges in such system as tuples composed of the message content and the vector of integers (representing the clock) at the time at which the message was emitted: $\langle content, clk_{msg}[1..n] \rangle$

### 3.4.4 Clock update protocol

The Vector clock specifies that when the node processes a new event the local logical clock (element i in the node i) must be increased by 1: $clk_{vect_i}[i] = clk_{vect_i}[i] + 1$.

Additionally, when a new event is processed the clock vector (representing the nodes view of the global logical time) is updated as follows [33]:

1. $\forall k$ s.t. $1 \leq k \leq n$: $clk_{vect_i}[k] = max(clk_{vect_i}[k], clk_{msg}[k])$: Each element of the vector clock is updated with the maximum value between the corresponding element received along with the message and the current value of the same element.
2. $clk_{vect_i}[i] = clk_{vect_i}[i] + 1$
3. Ready to process more events, including sending or receiving messages.

The CAST implementation follows:

```
//inside the actor we have definitions for:
// - num_nodes: number of nodes in the system
// - current_node_index: index of the current node
override def receiveCommand: Receive = {
case event =>
for (int k=0;i<num_nodes;k++) {
state.clk_vect[k] = max(state.clk_vect[k], event.c_msg[k])
}

state.clk_vect[current_node_index] = state.clk_vect[current_node_index] + 1;
processEvent(event);
}
```

The receiveCommand function (present in each Actor) is invoked by the Akka framework upon receipt of a new event, the state vector representing the clock is updated according to the clock specification (described above) and then the event is handled by the processEvent function.

### 3.4.5 Comparing Vector Clocks

Vector clocks provide the clock strong consistency property outlined in Definition 3.2.1, this means that by looking at the clock vectors of two separate events, we can determine if these are casually dependent one on the other. Comparing Vector Clocks requires establishing two main relations for comparing vector clocks [33]:

- $e_i \rightarrow e_j \iff \forall k$ s.t. $1 \leq k \leq n$: $\mathrm{CLK}(\mathrm{e}_i)[k] \leq CLK(e_j)[k]$.
- $\mathrm{e}_i \parallel e_j \iff \exists k, w$ s.t. $1 \leq k, w \leq n$: $CLK(e_i)[k] > CLK(e_j)[k] \wedge CLK(e_i)[w] < CLK(e_j)[w]$. (It is not possible to establish a total order between these events, therefore these events happen independently.)

By providing the clock strong consistency property Vector Clocks enable a more detailed understanding of time in distributed systems by making it possible to distinguish concurrent events from causal-dependent ones. Let us go back to the example in Figure Figure 1 and show how the introduction of Vector Clocks addresses the shortcomings highlighted above:

1. Both node1 and node2 start with $clk_{vect}$: [0, 0].
2. node1 processes a new event and the first element of the vector clock corresponding to the clock on node1 is incremented by 1: [1, 0].

3. node2 processes a new event and the second element of the vector clock corresponding to the clock on node2 is incremented by 1: [0, 1].
4. node1 wants to send a message to node2. It updates $clk_{vect}$ ([2, 0]) and then sends the message.
5. node2 receives a message from node1: it computes the new Vector Clock by applying the clock update protocol (As described in section 3.4.4): [2, 1].
6. node2 increments the second element of its vector clock by 1 and then processes the event it has received: [2, 2].
7. node1 processes a new event and the first element of the vector clock corresponding to the clock on node1 is incremented by 1: [3, 0].
8. node2 processes a new event and the second element of the vector clock corresponding to the clock on node2 is incremented by 1: [2, 3].

As we can observe, the third event of node1 and the second event of node2 don't share the same clock value anymore. The values are respectively [3, 0] and [2, 2]. More importantly, the use of the Lamport clock imposed a fictitious order in the example above: the third event on node1 had a lower associated clock value than the third event on node2, but the "happens before" relation did not hold. Instead, the two Vector clocks are now [3, 0] and [2, 3] respectively and by applying the clock strong consistency property outlined in Section 3.4.5 it is not possible to establish a total order between them. The two events are therefore independent.

### 3.5 Summary

In this chapter we have discussed the importance of the global clock in distributed systems in which very node exists in its own memory space and performs read and write operations on shared resources. We have shown the main steps we have taken in CAST to implement global clocks including defining data structures in every node and implementing a protocol to update these data structures in a way that keeps the clock consistent (Definition 3.0.6). Then, we have introduced two types of clocks: the Lamport clock and the Vector clock. We have indicated how they can be enabled in CASTDSL and used in CAST and we have described the CAST implementation details for each of them. Finally, we have highlighted the drawbacks of the Lamport clock and how the Vector clock successfully resolves such problems eliminating the fictitious order imposed by the Lamport clock on events and guaranteeing the clock strong consistency property.

# CHAPTER 4

# CONSISTENCY AND REPLICATION

In the previous chapter we have defined what a distributed system is and we have introduced the concepts of Events (Def. 3.0.4) and Clocks. In this chapter we will focus on how CAST implements an essential feature of distributed systems: replication. We will show how replication improves two key aspects of distributed systems: reliability and performance, how Logical Clocks (Def. 3.0.6) can be used to reconcile replicas and the multiple approaches to replication that CAST implements.

In traditional applications data is stored on a persistent storage medium (e.g. Hard Drive (HDD)) and then accessed through read/write operations. All data is stored in a single location, where the server is, and all clients access the central storage location. If the storage medium (e.g. Hard Drive (HDD)) fails, data becomes not accessible; if a client is in a different geographical location wrt the storage location it will incur in significant latency.

**Definition 4.0.1** (Latency)**.** Latency is a term that designates the period of delay when one component of a computing system is waiting for an action to be executed by another component. [36]

*Example* 4.0.1. A client is located in Melbourne, Australia and attempts to access the data contained in a website hosted in Europe. The data access operation is significantly slower compared to the same operation performed by a European user due to latency. This is because of the distance between the storage location and the user.

**Definition 4.0.2** (Logical data item)**.** A Logical data item, or data item, is the unit of information that can be manipulated through the execution of operations in the distributed system. A logical data item $i$ is a tuple: $\langle id, C \rangle$ where $id$ is a logical identifier and $C$ is a set of Replicas (Def. 4.0.3) s.t. $\mathrm{C} \coloneqq \{\, c \mid \mathrm{rid}(c) = \mathrm{id}(i) \,\}$

**Definition 4.0.3** (Replica)**.** In CAST, replicas are tuples $\langle id, Rid, anyMessage \rangle$ where $id, Rid$ are logical identifiers and $anyMessage$ is a message. Each replica is stored inside the state of a graph node. (Def. 4.0.2)

**Definition 4.0.4** (Replication transparency)**.** Replication transparency states that clients should not be aware that multiple copies (or replicas) of the data exist.

In the context of replication we distinguish Logical data items (Def. 4.0.2) from Replicas (Def. 4.0.3).

**Definition 4.0.5** (Reliability)**.** Reliability(x) is the probability that a cloud component has been up and running continuously in the time interval [0,x). [37]

Replication improves reliability in case of node failures[1] since the remaining nodes with replicas can continue processing events (Def. 3.0.4).

---

[1] **Node failure**: A node is said to have failed if it becomes unresponsive, returns unexpected messages or stops, totally or partially, receiving or sending messages. Please see section 6

In distributed systems we introduce the concept of replication or replicas with the objective of improving reliability and performance.

*Example* 4.0.2 (Reliability through replication)*.* For example in a distributed database server (See 3.0.1) the same logical data item $i$ is replicated on a subset $M \coloneqq \{m1, m2, m3\}$ of the nodes on which the database server runs resulting in the respective replicas: $R \coloneqq \{r1, r2, r3\}$. If a node, such as m1, fails a copy of the logical data item $i$ could still be retrieved from the remaining nodes {m2, m3}, by performing a read operation of either replica {r2, r3}. [31]

*Example* 4.0.3 (Performance through replication)*.* For example in a distributed database server (See 3.0.1) the same logical data item $i$ is replicated on a subset $M \coloneqq \{m1, m2, m3\}$ of the nodes on which the database server runs resulting in the respective replicas: $R \coloneqq \{r1, r2, r3\}$. If a node in the subset $M$ can process a single message $x1$ in the amount of time X and we send 3 messages $(x1, x2, x3)$, it will take 3X. If we have multiple replicas, we could (best case scenario) send 1 message to each node in parallel and process the 3 message in X time. [31]

*Remark.* The examples above give an intuitive understanding of the benefits of replication in distributed systems and represents a best case scenario. In example 4.0.2, for the sake of simplicity, we omit to mention the problem of guaranteeing consistency and synchronization among the replicas. In example 4.0.3, for the sake of simplicity, we assume that the messages $x1, x2, x3$ can be fully executed in parallel.

## 4.1 The consistency problem

As we have shown replication can improve reliability and performance in a distributed system, but a new problem arises: ensuring consistency among replicas.

Performing a write operation on one of the replicas results in that replica being modified (Ref 3.0.2) and becoming different from the rest. Eventually[1], writes need to be executed on all replicas to ensure consistency. [31]

*Example* 4.1.1 (Consistency in replication). Let us consider an example in which a user wants to access a web page. Obtaining a copy of the web page requires connecting to the web server via an internet connection and download the page, which can take seconds. (depending on the connection speed). The web browser can cache the web page locally, this means creating a copy of the web page and storing it on the hard drive. If the user asks for the same web page, now it can be returned immediately with close to zero-latency 4.0.1. (improved performance). Furthermore, if the web server becomes unavailable, the user can still view the local copy of the web page. (improved reliability). However, if we don't take any special action and the web page is updated on the server, the user will still see the local copy which is now stale or outdated. We need to make sure that the local copy and the online copy (called replicas) are always the same or at least synced up at some point in time. [31]

In the rest of this section we will introduce various algorithms and their implementations in CAST with the aim of ensuring consistency and proving replication.

## 4.2 Implementation framework for Replication

In this section we will introduce the steps necessary to implement replication in cloud simulators such as CAST. We will later describe how these steps are mapped to the CAST code.

---

[1] Please see section 4.2.

**Definition 4.2.1** (Group membership algorithm)**.** In CAST by group membership algorithm we mean an algorithm to group a set of nodes together, with properties associated to the group and properties associated to all group members.

Implementing replication requires:

1. Implementing a group membership algorithm. way to group a set of nodes together, with properties associated to the group and properties associated to all group members.
2. Implementing one or more algorithms to elect a master node for any logical data item.[1]
3. Choosing the data replication granularity.
4. Implementing a replica update propagation strategy (eager/lazy propagation)
5. Implementing a replica synchronization strategy which aims at solving conflicts between replicas. (replica reconciliation strategy)

**Definition 4.2.2** (Simple master replication model)**.** In the Simple master replication model, a master is elected among the nodes of the group for every logical data item (Def. 4.0.2) and all updates are submitted to the master node before being propagated to the other nodes. [38]

The CAST implementation of replication is based on the single master replication model. [38]. In this replication model updates are submitted in the same order to masters and respective replicas and replicas may become stale in between synchronization intervals.

---

[1] Election of a master node: The algorithms used to elect a master node are described in Section **??**.

## 4.3 Responsibilities of CAST

When replication is enabled for a logical data item, CAST takes the following steps:

1. Elects a node inside the group as master for that logical data item.
2. Creates a route for messages so that messages representing operations on a logical data item are routed to the master for that logical data item.
3. Creates the necessary data structures and enables the protocols of the clock chosen for replica reconciliation and synchronization. (As described in section **??**).
4. Performs synchronization periodically (replica reconciliation) using the clock.

## 4.4 Responsibilities of the SimEng

The SimEng responsibilities are:

- Specify which nodes belong to the group on which we want to enable replication.
- Specify the election algorithm or specify a list of master nodes for any data item type.
- Select the data replication granularity.
- Specify the synchronization interval and strategy.

## 4.5 Grouping nodes

A distributed system is a graph where each node is a computing entity. Nodes can be grouped together when the SimEng wants to enable a set of features for more than one node at once such as replication.

**Definition 4.5.1** (Group of nodes)**.** A Group of nodes is a set $G_{p,v} \coloneqq \{\, n \mid p \in parameters(n) \wedge val(p) = v \,\}$ where $p$ is a group parameter, $v$ is its value, $parameters(n)$ in a function returning the parameters of node $n$ and $value(p)$ is a function returning the value of a parameter $p$. In CAST groups of nodes are sets of nodes that share some common parameter $p$. For randomly chosen nodes the only common parameter $p$ is the stored data and its replicas.

*Remark.* Note that at least one common $p$ should exist in order for nodes to form a group.

In CASTDSL the SimEng can create groups by specifying the following construct [35]:

Listing 4.1: Sample Group construct.

```
groups:
    my_group:
        members: [node_0, node_1]
        group_parameters:
            election:
                type: ElectionAlgorithm
                value: BullyAlgorithm
            replication:
                type: SingleMaster
                replicated_items: [Movie]
                clock: ScalarLamport
        all_node_parameters:
            clock_type:
                type: ClockType
                value: ScalarLamport

```

The following are the steps taken by CAST when a ”group” tag in parsed in CASTDSL:

1. A new Actor is created with the same id as the group name. (”my_group”)

2. All messages with a destination equal to the id of any node in the group are routed to the Actor created previously.
3. The functionalities specified in the "group_parameters" are enabled on the "my_group" Actor created at step 1.
4. The functionalities specified in the "all_node_parameters" are enabled on all nodes, represented by Actors, belonging to the group.

*Remark* (Enabling a functionality). How a specific functionality is enabled is detailed in its respective section. For example Lamport clocks are enabled is described in section 3.2.

More details about the CAST architecture definition, including the definition of groups, are referenced from the "Architecture definition" section [35].

### 4.5.1 Logical ordering of nodes in a group

In CAST nodes that belong to a group can be logically ordered, this is useful for algorithms such as the Bully algorithm (Chapter 5) which require a total node order. In CAST, by default, the order of the nodes in the group is determined by the order in which the nodes appear in the "members" list (See group construct 4.1). Each node id (nid) is mapped to the node index in the list: $index \longrightarrow nid$ and stored in the Actor representing the node group.

## 4.6 Election of a master node

The CAST implementation of replication is based on the single master replication model [38] this means that CAST will elect a master node for each logical data item the SimEng wants to replicate. The SimEng has the responsibility of choosing the election algorithm he wants to use. CAST will apply the chosen election algorithm for each replicated logical data item (ref. 4.0.2) until a master node has been selected for each of them.

## 4.7 Replica reconciliation

As detailed in section 3.2 certain replicas become different from the rest when they are modified by write operations. This causes a consistency problem and makes it necessary to perform write operations on all replicas. In this section we will show how we can rely on logical clocks to achieve this goal.

**Definition 4.7.1** (Replica reconciliation)**.** Replica reconciliation is the process of propagating the updates introduced in a replica to the other replicas belonging to the replication group such that: [39]

- Perform updates (write operations) on all replicas in the same order.
- Propagate writes between replicas. If a write operation is executed on one replica, it must be reflected on the other replicas as well.

As described in the Logical Clocks section (ref. 3.2) we formalize the messages exchanges in a distributed system as tuples composed by the message content and the clock value at the time at which the message was emitted: $\langle content, clk_{msg} \rangle$ This operation is described as "tagging" a message with the clock value and consists in the association of a message with the clock value of the node where the message originated. [40]

*Example* 4.7.1 (Message and clock tagging). On a given node (A) the lamport logical clock time is 10 when message SampleMessage is created and is tagged with a clock value of 10. $\langle SampleMessage, 10\rangle$

In order to reconcile replicas we make use of Clocks as introduced in section 3.2. Two steps need to be accomplished:

1. "tag" each message with the respective clock value.
2. Use a replica reconciliation protocol based on the clock value.

### 4.7.1 Replica reconciliation protocol

CAST implements the Totally-Ordered Multicast protocol in order to achieve replica reconciliation. Intuitively all incoming messages are put in a queue and sorted by clock value so that all replicas apply the updates (write operations) in the same order as specified by the Lamport Logical Clock.

**Definition 4.7.2** (Client node). Given a set of nodes $G_{p,v}$ and the set $R$ of replicas (Ref. 4.0.2 4.5.1) such that $R$ are replicated on group $G_{p,v}$, a Client node is a node that is outside of group $G_{p,v}$ where no replicas belonging to set $R$ are present. Concisely, a node is said to be a Client node for a replication group if it is a node that is not part of the replication group.

**Definition 4.7.3** (Replication Message). A Replication Message is a tuple $\langle ack, content, clk_{msg}\rangle$ where $ack$ is a boolean representing whether this message was acknowledged, content is the message content and $clk_{msg}$ is the clock value (an integer) at the time at which the message was emitted. anyMessage(s) are translated into Replication Messages as shown: $\langle id, content, clk_{msg}\rangle \longrightarrow \langle id, 0, content, clk_{msg}\rangle$.

In the rest of this section the protocol used by CAST to reconcile replicas is described in detail [40]:

1. Any time a new anyMessage is received from a client node translate it into a ReplicationMessage and send it to all nodes part of the replication group. (including the node that received the anyMessage or current node)
2. Any time a new ReplicationMessage is received by a node from another replica, the node must:
   (a) Add the message to the node's local queue
   (b) Send an acknowledgement message to all other replicas
3. Any time a new acknowledgement message is received by a node, the node must:
   (a) Mark the corresponding message as acknowledged in the node's local queue
4. Any time a ReplicationMessage has been acknowledged by all replicas, it must be removed from the queue and processed.

## 4.8 Summary

In this chapter we have presented two key aspects of distributed systems, reliability and performance, and we have focused on how CAST implements as essential feature of distributed systems: replication. We have introduced the concepts of latency and replicas along with appropriate examples for the concepts of reliability and performance in distributed systems. We have discussed the problem of consistency which arises when a write operation on a replica results in that replica being modified and becoming different from the other replicas. Additionally, we have explained how nodes of the distributed system can be grouped together and how CAST manages replication for these groups of nodes. Finally, we have described the problem of replica reconciliation and how it can be achieved by relying on logical clocks introduced in Chapter 3 and the specific replica reconciliation protocol implemented in CAST.

# CHAPTER 5

# ELECTION ALGORITHMS

A distributed system is a graph where each node is a computing entity. Nodes in the graph can form groups and run distributed algorithms[1] to execute read and write operations.

Distributed algorithms allow for the sequence of operations to be split among multiple nodes and some require a node to act as a Coordinator. [31] This special node is responsible for organizing the execution of the distributed algorithm and communicates with the other nodes in the group to schedule operations or collect results [2].

*Example* 5.0.1 (Distributed Commit Algorithms and Coordinators). A family of algorithms which require a Coordinator node are the Distributed commit algorithms, such algorithms coordinate whether nodes that take part in a distributed transaction should commit a transaction or abort it. An example is the two-phase commit protocol (2PC protocol), widely-used in distributed systems [31].

For example the 2PC protocol requires the election of a Coordinator node which is the only node responsible for [31]:

---

[1] Distributed algorithm: A distributed algorithm is an algorithm that runs on more than one node.

[2] Examples of the responsibilities of the coordinator which depend on the specific distributed algorithm.

1. Sending a VOTE_REQUEST message to all the nodes in the group to query them about whether they are prepared to commit the transaction locally.
2. Collecting a response (VOTE_COMMIT, VOTE_ABORT) from all the group nodes indicating whether they are ready to commit the operation locally.
3. Sending a message to all group nodes confirming or aborting the commit operation.

While all the remaining nodes in the group reply to the messages sent by the Coordinator and act accordingly.

As it can be noticed from the example above, algorithms which require a Coordinator don't prefer a specific node over another in the group.

*Remark.* If the results produced by a given algorithm do not change depending on the node chosen as Coordinator we can use Election algorithms to select a Coordinator among the nodes in the group. [31]

In the rest of this section we will show how election algorithms can be used to elect a node as Coordinator among the nodes in a group, the steps necessary to implement election algorithms including the definition of the message format and APIs used in CAST and, finally, we will introduce some common election algorithms implemented in CAST.

*Remark.* In order to elect one node of the group as the Coordinator we need to be able to distinguish each node in the group. In CAST we assume that each node in the group is uniquely identified by a node *id*. This assumption is realistic and the *id* field corresponds to the network address of the node in the real world. Additionally, we assume that:

- Every node knows which are the other nodes in the group.
- Nodes in the group are not aware of whether other nodes which belong to the same group have failed or not.

### 5.0.1 Goal of an election algorithm

The goal of an election algorithm is to guarantee that once an election[1] starts, a new Coordinator is elected and all nodes in the group use the newly elected node as their new Coordinator. [31]

## 5.1 Implementation framework for Election Algorithms

In this section we will introduce the steps necessary to implement election algorithms in CAST.

*Remark.* The CAST run-time knows that a Coordinator must be elected when the current Coordinator has failed or no Coordinator has been chosen yet. More details on how to determine a node has failed and the failure models available in CAST are contained in section 6.

Implementing Election algorithms in CAST means solving the following issues:

- Providing a way for a node to signal the start of the election process to the other nodes in the group.
- Provide a way for nodes in the group to respond to the election start signal.
- Provide a way for a node in the group to notify other nodes that a certain node (including itself) is the new Coordinator.

In order to implement an election algorithm we need to:

---

[1] Election: an election is the process of choosing a new Coordinator among the nodes in a group.

- Define a set of election messages.
- Define the election APIs made available to SimEngs.

## 5.2 Definition of election messages

Based on the issues outlined above we identify three main categories of messages for implementing election algorithms in CAST:

- Election Message ($ElectionMessage$ object): Message sent from a node $n1$ in the group to any other node when $n1$ wants to start the election process.
- Node Status Message ($NodeStatusMessage$ object): Message sent from a node in the group to any other node that carries the node status and/or additional information about the sending node. This message is used to query a node status given assumption 5.
- Coordinator Message ($CoordinatorMessage$ object): Message sent from a node in the group to all other nodes in the group to indicate the sending node is taking over as a Coordinator.

### 5.2.1 CAST Messages for Election

In CAST, we formalize Election Messages as tuples composed by the sender of the message, the destination actor and a data field $\langle sender, dest, data\rangle$.

### 5.2.2 Message Sequence in Election algorithms

Election algorithms differ one another, CAST provides some election algorithms out-of-the-box in addition to allowing SimEngs to define their own. Although Election algorithms are different, the following is the common message sequence in terms of CAST election messages:

1. A node in the group sends an *ElectionMessage* to any other node requesting an election to start.
2. The receiver node of the *ElectionMessage* responds by sending a *NodeStatusMessage* indicating the node status to the sender node.
3. Once, the algorithm produces the new Coordinator node a *CoordinatorMessage* is circulated among the nodes of the group to notify them of the Coordinator change.

## 5.3 Definition of election APIs

In addition to definiting election message (Section 5.2), CAST defines Election APIs that SimEngs can use to obtain information about elections.

The following APIs can be invoked from inside each actor node in a group for which election is enabled. (See section 5.6 for enabling Election algorithms in CAST):

- *getCoordinatorId*(): Obtain the identifier *id* of the current Coordinator node for the group.
- *sendMessageToCoordinator*(*message*): Send a message to the current Coordinator for the group previously selected by an election.

## 5.4 Functionality of CAST

When election is enabled for a group of nodes, CAST takes the following steps:

- Uses a failure model specified for the group of nodes (See section 6) to determine whether the Coordinator has failed.

- Creates the necessary data structures to keep track of the current Coordinator node.
- Creates a route for messages so that all messages sent with the sendMessageToCoordinator(message) API are sent to the current Coordinator node.

*Remark* (Coordinator failure transparency). In CAST each time the API sendMessageToCoordinator(message) is invoked, the status of the Coordinator is checked. If the Coordinator is known to have failed, then the message waiting to be sent to the Coordinator is persisted, the election process takes place and the pending message is sent to the new Coordinator transparently.

## 5.5 Responsibilities of the SimEng

The SimEng responsibilities are the following:

- Specify which nodes belong to the group on which we want to enable an election algorithm.
- Specify the election algorithm among the out-of-the-box algorithms offered by CAST via the ”election” value field in CASTDSL.
- Using the $sendMessageToCoordinator(message)$ to send messages to the elected Coordinator.

## 5.6 Enabling Election algorithms in CAST

In CAST the chosen election algorithm is a group property, this means that the chosen election algorithm applies to a group of nodes. Enabling Election algorithms in CAST is done by SimEngs by specifying the election group property in CASTDSL as already shown in section 4.5. In particular election is enabled as follows:

Listing 5.1: Enabling election in CAST.

```
groups:
    my_group:
        members: [node_0, node_1]
        group_parameters:
            election:
                type: ElectionAlgorithm
                value: Bully
```

. The valid values of the election value field are specified in Appendix **??**.

## 5.7 Bully algorithm

In the following section we will present the bully algorithm that CAST implements for election of a Coordinator among the nodes in a group.

*Remark.* For the purposes of implementing the algorithm, we assume that a total order of the node ids can be established. That is, each node knows how to compute the id of the successor of a node. (See section 4.5.1)

When any node ($n1$) detects that the Coordinator has failed or if no Coordinator has been chosen yet, it takes the following steps [31]:

1. n1 sends an $ElectionMessage$ to all nodes with a higher id than the id of n1. (n1 has detected the Coordinator failure)

2. If no response is detected, n1 wins the election and becomes the new Coordinator. n1 sends a $CoordinatorMessage$ to all other nodes in the group to signal it has acquired the role of Coordinator.
3. If any other node responds with a $NodeStatusMessage$, n1 terminates the election process and the other node takes over and starts the process again from step 1.

In detail [31]:

- When a node receives an $ElectionMessage$ from any node, the receiving node responds by sending a $NodeStatusMessage$ if the id of the sender of the message is lower than the id of the node.
- When a node receives a $NodeStatusMessage$ it means that a node with a higher id is alive[1] and it will take over the election process.
- In the end all nodes except the one with the highest id terminate the election process following a $NodeStatusMessage$ reply from another node. The highest id node wins the election and becomes the new Coordinator.

### 5.8 A ring algorithm

In the following section we will present a ring algorithm that CAST implements for election of a Coordinator among the nodes in a group.

---

[1] Alive: Not failed.

*Remark.* For the purposes of implementing the algorithm, we assume that a total order of the node ids can be established. That is, each node knows how to compute the id of the successor of a node. Furthermore, the nodes are organized in a ring, the successor of the node with the highest id in the group is the node with the lowest id: $successor.nN = n1$ where $N$ is the number of nodes in the group.

When any node ($n1$) detects that the Coordinator has failed or if no Coordinator has been chosen yet, it takes the following steps [31]:

1. n1 sends an $ElectionMessage$ containing a list of length 1 with its node id as the only element (data element of the message tuple as defined in section 5.2.1) to its successor node (n1 has detected the Coordinator failure)
2. The successor responds with a $NodeStatusMessage$ and repeats step 1 adding its own node id in the $ElectionMessage$.
3. If no response is detected, n1 sends an $ElectionMessage$ to the successor of its successor and so on until it gets a $NodeStatusMessage$ response.
4. This process continues until a node receives an $ElectionMessage$ with a list containing its own node id. This means the $ElectionMessage$ has reaches all nodes in the ring.
5. The $ElectionMessage$ is converted to a $CoordinatorMessage$ containing the id of the highest node in the list of node ids and is sent around the ring as it happened for the previous $ElectionMessage$.

## 5.9 Summary

In this chapter we have discussed election algorithms and we have shown that nodes in the graph can form groups and run distributed algorithms which require a node in the group to act as a Coordinator. [31] Among the available algorithms to elect a node among the members of the group we have presented the Bully algorithm (Section 5.7) and a ring algorithm (Section 5.8). Finally, we have discussed their implementation in CAST in terms of messages and of APIs offered to SimEngs.

# CHAPTER 6

# PARTITIONING AND FAILURE MODELING

In the previous chapters we have introduced the concepts of Events (Def. 3.0.4), Clocks, Replication (Chapter 4.2) and Election (Chapter 5). In this chapter we will focus on failures, which we first introduced while discussing the concept of reliability in Section 4.0.5. We will define what failures are in distributed systems, which are the different types of failures and how failures are modeled in CAST.

**Definition 6.0.1** (Failure in Distributed Systems)**.** A distributed system has failed if if one or more services the system provides to its users can not be - in full or partially - provided. [31]

**Definition 6.0.2** (Partial failure)**.** In a distributed system a partial failure is the failure of a subset of components[1] of the distributed system. [31]

A distinguishing feature of distributed systems is the concept of partial failure, where only some components of the system can fail leaving the remaining ones untouched while in traditional non-distributed systems failures affect the entire system [31].

*Remark* (Failures in CAST)*.* In CAST we model distributed systems as graphs where each node is a computing entity, the components of the system subject to failures are modeled as nodes in the graph. In CAST nodes can represent network channels as well.

---

[1] In CAST by component we mean any node in the graph representing the distributed system.

## 6.1 Failure types

In this section we will present the most common failure types that can happen in distributed systems that are modeled in CAST. The classification of failures throughout this work is based on the work described in Cristian 1991 [41] and Hadzilacos and Toueg 1993 [42] [31]. Failures in distributed systems are divided in [43] [31]:

- Crash failures.
- Timing failures.
- Response failures.
- Arbitrary failures.

### 6.1.1 Crash failures

**Definition 6.1.1** (Crash failure)**.** A crash failure is a type of failure in which a node stops working at a certain time instant $t$, but it was working correctly[1] until time instant $t$. After $t$ the node becomes unreachable and no further messages are received from it.

*Example* 6.1.1. Let us consider a distributed system made of two nodes which handle incoming requests from users. One of the two nodes runs out of available memory (RAM) and the operating system stops working. As a consequence the node stops working at a certain time $t$ and will no longer work unless restarted. Before running out of available memory the node was working correctly and responding to user requests.

---

[1] Working correctly: Not failed.

### 6.1.2 Timing failures

**Definition 6.1.2** (Timing failure)**.** A timing failure is a type of failure in which a response message from a node arrives outside a specified period of time. Given a time interval $[0, t_{max}]$ representing the maximum length of time any node will wait for a reply to a previously sent message, a timing failure occurs if the reply arrives at any time instant $t$ s.t. $t > t_{max}$.

*Example* 6.1.2. Let us consider a node that receives requests for issuing One Time Password (OTP) tokens for web-applications valid for 60 seconds. Another node requests a new token at time $t_0$. The token is generated on the server at $t_1$ and is valid until $t_{61}$. In order for the reply of the server to be useful, the node requesting the token must receive a reply before $t_{61}$. If the node receives the reply at $t > t_{61}$ a timing failure occurs. In general a timing failure occurs any time a reply message is received outside a certain validity period.

### 6.1.3 Response failures

**Definition 6.1.3** (Response failure)**.** A response failure is a type of failure in which a response message from a node is systematically incorrect.

*Example* 6.1.3. Let us consider a node of a distributed system used to query a database and return results. If a node is sent a SELECT query with a WHERE clause on a given column $c$ asking it to return all tuples for which $c$ has a certain value $v1$, but instead returns tuples with $c$ equal to $v2$, then this node is experiencing a response failure. The node is returning an incorrect response to the query performed.

### 6.1.4 Byzantine failures

**Definition 6.1.4** (Byzantine failure)**.** A byzantine failure is a type of failure in which different nodes of the distributed system have a conflicting view of reality. [31] [44].

*Example* 6.1.4. Let us consider three nodes $n_x$, $n_y$ and $n_z$. $n_x$ and $n_y$ are connected through a working network link, $n_y$ and $n_z$ are connected through a working network link as well while the network link between $n_z$ and $n_x$ is faulty. In this scenario the node $n_y$ believes that $n_z$ is functioning correctly, while $n_x$ believes that $n_z$ has failed because it appears unreachable. Each node $n_x$, $n_y$ belies that the information that it has at its disposal is accurate. Let us assume that $n_z$ is the coordinator (see Section 5), node $n_x$ will initiate an election process since the coordinator is unreachable, but will receive a conflicting response from node $n_y$ that is still communicating with the old coordinator and does not see a reason to start an election process. There is imperfect information as to if $n_x$ has failed. [44]

## 6.2 Implementation framework for Failures

In this section we will introduce a framework to model the failures described in the previous section in CAST and provide support for failures in the simulation. We will later describe how this framework is mapped to the CAST code for each type of failure we consider.

*Remark* (Nodes in CAST)*.* In CAST failures are defined as node failures. In other words only the nodes in the graph can fail. It is important to remind the reader that nodes can represent any component of the distributed system, including network links and therefore this does not impose any limitation. Please see *M. Pelosi, "CAST: Cloud-based applications simulator testbed."* for details on how network links are modeled as nodes in CAST [35].

Implementing support for failures in the simulation requires:

- Being able to generate failures during the simulation, either at a specific time instant or following a certain probabilistic distribution which reflects failure distribution in the real distributed system.
- Providing a way for the SimEng to handle the failure and run custom code inside each node when the failure happens.
- Providing a way for nodes in the graph to recover from failures and return to a non-failed state.

More in detail, CAST should:

- Implement a CASTDSL construct for failure generators.
- Support the execution of custom code in response to a failure of a given type by invoking the respective handler inside each actor representing a node of the graph.
- Provide out-of-the-box support to allow SimEngs to send recovery messages to actors which in response should switch to a non-failed state.

### 6.2.1 Responsibilities of the SimEng

The SimEng responsibilities are:

- Creating a failure generator by specifying the failure type among the failure types offered by CAST along with failure distribution and distribution parameters.
- Send a message to start/stop the failure generator using the simulation workflow messages.
- Send recovery messages to actors.

### 6.2.2 Failure generators

In this section we describe failure generators in CAST as previously introduced in Section 6.2. When the SimEng wants to create a failure generator as part of the simulation, he needs to specify the following CASTDSL construct. The construct appears in the root of the yaml file and should not be nested in any other construct.

Listing 6.1: Sample Failure Generator construct.

```yaml
failure_gens:
    failure1:
        group: group_1
        failure_type: CrashFailure
        distribution: poisson
        distribution_params:
            name: lambda
            value: 3
    failure2:
        group: group_2
        failure_type: TimingFailure
        distribution: poisson
        distribution_params:
            name: lambda
            value: 3
```

The construct above contains:

- Failure generator name such as ”failure1”,”failure2”. Must be unique.
- **group**: group of nodes (See Section 4.5) to which this failure generator applies.

- **failure_type**: The type of failure for this generator among the types listed in Section 6.1. Valid keywords are listed in Appendix **??**.
- **distribution**: Probabilities distribution from which samples will be drawn.
- **distribution_params**: Parameters of the chosen distribution.

The following are the steps taken by CAST when a "failure_gens" tag is parsed in CASTDSL. For each of the children of the "failure_gens" tag, each representing a failure generator name:

1. A new Actor is created with the same id as the failure generator name. ("failure1","failure2")
2. The Actor state is updated to generate failures of type "failure_type" and using distribution "distribution".
3. All messages generated by the Actor are sent to the Actor with the same id as the group specified in the field "group". (See Section 4.5)

*Remark.* A similar approach to failure generators is present in *M. Pelosi, "CAST: Cloud-based applications simulator testbed."* for generating simulation inputs. [35]

*Remark.* CAST stores the "failure_type", "distribution" and "distribution_params" values inside the node state of the Actor representing the failure generator. Each different "failure_type" value activates a specific logic contained in the Actor code which we describe in the following sections.

### 6.2.3 Failure generators integration

In CAST distributed systems are represented as a graph where each node is a computing entity, nodes in the graph are organised in groups. Each group is represented by the corresponding node Actor with the same name of the group (Group Actor) as described in detail in Section 4.5. This Actor receives all the messages for any of the group members (nodes) and forwards them to the correct member. Also, all outgoing messages from any group member are routed through this Actor before reaching their destination.

In CAST failure generators apply to group of nodes. The following communication takes place between the failure generator and the Group Actor:

1. A failure generator is linked to a group via the ”group” property of the CASTDSL construct.
2. The Group Actor, instead of forwarding messages directly to the group, forwards all messages to the failure generator.
3. The failure generator transforms the messages received and returns them to the Group Actor for delivery.

### 6.2.4 Responding to failures

In this section we describe how CAST allows SimEngs to respond to failures during a simulation as introduced in Section 6.2.

CAST supports the execution of custom code in response to a failure and allows SimEngs to extend the nodes functionality to handle the failure. When a failure event occurs, CAST:

- The failure message is routed to the node member by the Group Actor (Section 4.1).
- The Group Actor stops routing any message to the node.
- CAST invokes the failure handler so the SimEng can handle the failure from inside the actor code.
- The Group Actor resumes routing of messages once the handler code has finished executing.

override def receiveCommand: Receive = case CrashFailure =¿ The receiveCommand function (present in each Actor) is invoked by the Akka framework upon receipt of a new failure event, the custom code is executed and then the actor waits for the next messages to the received.

### 6.2.5 Failure control messages

As described in the previous sections failure generators are able to produce failures during a CAST simulation based on samples drawn from a statistical distributions. [45] In other to coordinate and control failure generators, CAST defines a set of control messages that the SimEng can send to failure generators to control their behaviour. [35] We identify the following messages for sending control signals to failure generators in CAST:

- Start Message ($FailureStartMessage$ object): Message sent to a failure generator indicating that we want that generator to start producing failures based on samples drawn from the statistical distribution.

- Stop Message ($FailureStopMessage$ object): Message sent to a failure generator indicating that we want that generator to stop producing failures based on samples drawn from the statistical distribution.
- Node recover Message ($FailureRecoverMessage$ object): Message sent to any node in a group indicating that that node should return to a non-failed state after a failure occurred.[1]

---

[1] $FailureRecoverMessage$: This message is used only for Crash Failures.

## 6.3 Summary

In this chapter we have discussed the concept of failures in CAST. In particular, we have provided the definition of failure and introduced the concept of partial failures specifically relevant to distributed systems. We have identified four types of failures: crash failures, timing failures, response failures and arbitrary failures and we have highlighted their differences. In CAST we model failures as node failures, in other words only nodes in the graph representing the distributed system can fail. Then, we have described how we have implemented support for failures including being able to generate failures during the simulation, providing a way for SimEngs to handle failures by running custom code and providing recovery mechanisms to allow nodes to return to a non-failed state. Finally, we have detailed the failure generators implementation and the types of the failure control messages.

# CHAPTER 7

# RELATED WORK

With the constant growth of cloud computing it is increasingly important for researchers to understand and predict the behaviour of large scale cloud datacenters and to analyze novel distributed algorithms running on top of them. Large scale simulators are a key tool for evaluating distributed algorithms and understanding their behaviour. CAST, a Cloud-based Applications Simulator Testbed, makes large-scale simulations possible in a simple and intuitive way. The aim of this work is to integrate high-level functionalities such as the Global Clock, Consistency and Replication algorithms, Election Algorithms and Partitioning and Failure modeling into an underlying cloud simulator architecture while providing intuitive access to such functionalities even for engineers with limited programming experience. In this chapter we investigate the state-of-the-art of cloud simulators that we can use as the underlying architecture and, at the same time, we analyze the availability of implementations of algorithms related to Clocks, Consistency, Election algorithms and Partitioning and Failure modeling in the literature.

## 7.1 Cloud Simulators

In this section we investigate various cloud simulators present in the literature with the aim of choosing one of them as the underlying architecture for the algorithmic framework we have developed as part of this work. For each cloud simulator we consider we evaluate its ability to run large scale simulations, its extensibility and its ease of use for SimEngs unaware of the simulator internals.

### 7.1.1 CloudSim

CloudSim is an extensible simulation toolkit written in Java which allows users to run cloud simulations by defining a set of system components (Datacenters, VMs) and tasks or cloudlets that represent operations run on one of more system components. [46] In order to run simulations on top of Cloudsim, users need to import the simulation code in a new Java project and edit the Java code to configure the simulation specifics. Additionally, CloudSim instantiates a Java object for each system component the users wishes to use and provides no support for distributing objects across a set of physical machines. Work present in the literature attempts to solve this significant issue by creating a Java-based actor-model by relying on Hazelcast[47]. Users need to learn how the Cloudsim architecture is structured before they can run simulations. [12] For these reasons, we are not considering CloudSim as the underlying architecture for the algorithmic framework we have developed.

### 7.1.2 CAST: A distributed simulation framework

CAST, a Cloud-based Applications Simulator Testbed, is a novel simulation framework written in Scala and based upon Akka, an Actor-based framework. [48] Its use of the Actor model allows CAST not to instantiate a Java class for each cloud component the user creates, but instead, use lightweight actors with a significantly smaller memory footprint. Additionally, Akka provides support for distributing Actors across a cluster of physical machines by relying on "Akka Cluster Sharding" in a seamless and transparent way. CASTDSL, the CAST Declarative Specification

Language, provides users with an effective way to setup and run simulations only by relying on a declarative language with no specific knowledge of CAST architecture [35]. CAST is the underlying architecture we have chosen for this work.

## 7.2 Algorithmic implementations

In this section we analyze various algorithmic implementations of Clocks, Consistency and Replication, Election Algorithms and Partitioning and Failure modeling and their availability in existing cloud simulators.

### 7.2.1 Global Clock algorithms

In distributed systems messages are exchanged concurrently and the same resources are manipulated asynchronously by multiple nodes. When multiple messages access the same resource at the same time the order of such messages (and consequently of the operations) directly influences the output of the system. Clocks are the key to order the events in distributed systems and are an essential feature of distributed systems. However, clock algorithms, like the Lamport Clock and Vector Clock that we have presented as part of this work, are not present in most cloud simulators. CloudSim employs a simple Java counter to order operations during the simulation. [46] As we have discussed in Chapter 3 a simple timestamp, or counter, is insufficient for ordering events in distributed systems as it can introduce a fictitious order among events and generate collisions. CloudSim does not consider this issue because it does not allow the execution of concurrent operations, imposing therefore a behavioural restriction[1] of the cloud architecture.

### 7.2.2 Consistency and Replication

In distributed systems replication algorithms improve reliability and performance while guaranteeing consistency among replicas when a write operation on one of the replicas results in that replica being modified and becoming different from the rest. Cloud simulators, such as CloudSim don't implement any replication or consistency algorithm out of the box leaving it up to the users to provide their own replication implementation along with any support feature required. Simulators for consistency models, such as Herd [49], are available and aim at providing a theoretical memory consistency model simulator. Such simulators provide all the possible combinations of events allowed in a certain consistency model and are a useful tool to verify whether the chosen consistency constraints are enough for the user application [49]. This work takes a more practical approach with the focus on allowing the simulation of real-world architectures such as the Master-Slave architecture while guaranteeing consistency.

### 7.2.3 Election algorithms

Election algorithms are necessary to run some distributed algorithms which allow for a sequence of operations to be split among multiple nodes and require a Coordinator [31]. The Coordinator is responsible for organizing the execution of the distributed algorithm and communicates with the other nodes in the group to schedule operations or collect results. Election algorithms have

---

[1] Behavioural restriction: CloudSim sequences all events in the distributed system making them sequential.

been implemented in existing cloud simulators only for specific use-cases such as MapReduce simulations in Cloud2Sim [50] [51]. In these specific instances the simulator manages the Election in a transparent way. As part of this work we aim at providing some election algorithms such as the Bully algorithm or a ring algorithm and the building blocks to implement further election algorithms in an Actor-based architecture.

### 7.2.4 Partitioning and Failure Modeling

Failures, and in particular partial failures, are especially relevant to distributed systems: only some components of the system can fail leaving the remaining ones untouched while in traditional non-distributed systems failures affect the entire system [31]. This can lead to unexpected and complex behaviours which can be investigated through simulation [52]. Although not originally part of CloudSim, fault injection modules for CloudSim such as "FIM-SIM" are present in the literature. [45] "FIM-SIM" is a failure generator module which creates faults in CloudSim using samples drawn from statistical distributions such as Poisson and Weibull. The main drawbacks of such module is that it is tightly integrated with CloudSim and it is limited to failure generation. This work builds upon fault injection modules such as "FIM-SIM" by adapting them to work with the Actor model and, in addition to the failure generation capabilities, by adding additional APIs that allow SimEngs to respond to failures by executing custom code from within actors.

# APPENDICES

# Appendix A

## CASTDSL KEYWORDS

This appendix contains the list of CASTDSL keywords used throughout this thesis along with the feature they relate to and a description.

## Appendix A (continued)

| Keyword | Related feature | Description |
|---|---|---|
| ClockAlgorithm | Clock | Specifies the associated value is a Clock algorithm. |
| ScalarLamport | Clock | Scalar Lamport clock (3.2) |
| VectorClock | Clock | Vector clock (3.4) |
| ElectionAlgorithm | Election | Specifies the associated value is an Election algorithm. |
| BullyAlgorithm | Election | Bully election algorithm (5) |
| SingleMaster | Replication | Single Master replication |
| CrashFailure | Failures | Crash failure (6.1.1) |
| TimingFailure | Failures | Timing failure (6.1.2) |
| ResponseFailure | Failures | Response failure (6.1.3) |
| ByzantineFailure | Failures | Byzantine failure (6.1.4) |

TABLE III: CASTDSL Keywords to Feature

# CITED LITERATURE

1. Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, M.: A view of cloud computing. Commun. ACM, 53(4):50–58, April 2010.

2. Data centers: Jobs and opportunities in communities nationwide. Available at the US Chamber of Commerce - Technology Engagement Center: `https://www.uschamber.com/sites/default/files/ctec_datacenterrpt_lowres.pdf` (2020/01/17).

3. Gartner projects cloud services industry to grow exponentially through 2022. Available at Gartner, Inc.: `https://www.gartner.com/en/newsroom/press-releases/2019-04-02-gartner-forecasts-worldwide-public-cloud-revenue-to-g` (2020/01/17).

4. Filelis-Papadopoulos, C. K., Gravvanis, G. A., and Morrison, J. P.: Cloudlightning simulation and evaluation roadmap. In Proceedings of the 1st International Workshop on Next Generation of Cloud Architectures, CloudNG:17, pages 2:1–2:6, New York, NY, USA, 2017. ACM.

5. Paxson, V. and Floyd, S.: Why we don't know how to simulate the internet. In Proceedings of the 29th Conference on Winter Simulation, WSC '97, pages 1037–1044, Washington, DC, USA, 1997. IEEE Computer Society.

6. Li, B. H., Chai, X., Hou, B., Yang, C., Li, T., Lin, T., Zhang, Z., Zhang, Y., Zhu, W., and Zhao, Z.: Research and application on cloud simulation. In Proceedings of the 2013 Summer Computer Simulation Conference, SCSC '13, pages 34:1–34:14, Vista, CA, 2013. Society for Modeling &#38; Simulation International.

7. Liu, D., De Grande, R. E., and Boukerche, A.: Towards the design of an interoperable multicloud distributed simulation system. In Proceedings of the 50th Annual Simulation Symposium, ANSS '17, pages 13:1–13:12, San Diego, CA, USA, 2017. Society for Computer Simulation International.

8. Pan, Q., Pan, J., and Wang, C.: Simulation in cloud computing envrionment. In Proceedings of the 2013 International Conference on Service Sciences, ICSS '13, pages 107–112, Washington, DC, USA, 2013. IEEE Computer Society.

## CITED LITERATURE (continued)

9. Zehe, D., Cai, W., Knoll, A., and Aydt, H.: Tutorial on a modeling and simulation cloud service. In Proceedings of the 2015 Winter Simulation Conference, WSC '15, pages 103–114, Piscataway, NJ, USA, 2015. IEEE Press.

10. Fakhfakh, F., Kacem, H. H., and Kacem, A. H.: An evaluative review and research challenges of the simulation in cloud environment. Int. J. Softw. Innov., 5(4):59–73, October 2017.

11. Liu, X., Qiu, X., Chen, B., and Huang, K.: Cloud-based simulation: The state-of-the-art computer simulation paradigm. In Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation, PADS '12, pages 71–74, Washington, DC, USA, 2012. IEEE Computer Society.

12. Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A. F., and Buyya, R.: Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Softw. Pract. Exper., 41(1):23–50, January 2011.

13. Taylor, S. J. E., Khan, A., Morse, K. L., Tolk, A., Yilmaz, L., Zander, J., and Mosterman, P. J.: Grand challenges for modeling and simulation. Simulation, 91(7):648–665, July 2015.

14. Taylor, S. J. E., Khan, A., Morse, K. L., Tolk, A., Yilmaz, L., and Zander, J.: Grand challenges on the theory of modeling and simulation. In Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative M&S Symposium, DEVS 13, pages 34:1–34:8, San Diego, CA, USA, 2013. Society for Computer Simulation International.

15. Cayirci, E.: Modeling and simulation as a cloud service: A survey. In Proceedings of the 2013 Winter Simulation Conference: Simulation: Making Decisions in a Complex World, WSC '13, pages 389–400, Piscataway, NJ, USA, 2013. IEEE Press.

16. Byrne, J., Byrne, P., e Ferreira, D. C., and Ivers, A. M.: Towards a cloud based sme data adapter for simulation modelling. In Proceedings of the 2013 Winter Simulation Conference: Simulation: Making Decisions in a Complex World, WSC '13, pages 147–158, Piscataway, NJ, USA, 2013. IEEE Press.

17. Szufel, P., Czupryna, M., and Kamiński, B.: Optimal execution of large scale simulations in the cloud: The case of route-to-pa sim online preference simulation. In Proceedings

## CITED LITERATURE (continued)

of the 2016 Winter Simulation Conference, WSC '16, pages 3702–3703, Piscataway, NJ, USA, 2016. IEEE Press.

18. Johnson, H. E. and Tolk, A.: Evaluating the applicability of cloud computing enterprises in support of the next generation of modeling and simulation architectures. In Proceedings of the Military Modeling & Simulation Symposium, MMS '13, pages 4:1–4:8, San Diego, CA, USA, 2013. Society for Computer Simulation International.

19. Taylor, S. J. E., Kiss, T., Terstyanszky, G., Kacsuk, P., and Fantini, N.: Cloud computing for simulation in manufacturing and engineering: Introducing the cloudsme simulation platform. In Proceedings of the 2014 Annual Simulation Symposium, ANSS '14, pages 12:1–12:8, San Diego, CA, USA, 2014. Society for Computer Simulation International.

20. Li, Z., Chen, B., Liu, X., Ning, D., Duan, W., Qiu, X., and Xu, C.: Qos-aware parallel job scheduling framework for simulation execution as a service. In Proceedings of the 21st International Symposium on Distributed Simulation and Real Time Applications, DS-RT '17, pages 208–211, Piscataway, NJ, USA, 2017. IEEE Press.

21. Jones, A., Shao, G., and Riddick, F.: Enabling control system and cloud-based simulation service interoperability. In Proceedings of the 2018 Winter Simulation Conference, WSC '18, pages 703–714, Piscataway, NJ, USA, 2018. IEEE Press.

22. Li, F., LaiLi, Y., Zhang, L., Hu, X., and Zeigler, B. P.: Service composition and scheduling in cloud-based simulation environment. In Proceedings of the Model-driven Approaches for Simulation Engineering Symposium, Mod4Sim '18, pages 2:1–2:10, San Diego, CA, USA, 2018. Society for Computer Simulation International.

23. Madden, M. M. and Glaab, P. C.: Distributed simulation using dds and cloud computing. In Proceedings of the 50th Annual Simulation Symposium, ANSS '17, pages 3:1–3:12, San Diego, CA, USA, 2017. Society for Computer Simulation International.

24. Kacsuk, P.: Enabling distributed simulations using big data and clouds. In Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, SIGSIM PADS '15, pages 125–126, New York, NY, USA, 2015. ACM.

25. Borshchev, A. and Churkov, N.: Anylogic cloud: Cloud-based simulation analytics. In Proceedings of the 2018 Winter Simulation Conference, WSC '18, pages 4245–4245, Piscataway, NJ, USA, 2018. IEEE Press.

## CITED LITERATURE (continued)

26. Rajaei, H., Alotaibi, F., and Jamalian, S.: A dististributed simulation platform for cloud computing. In Proceedings of the 20th Communications & Networking Symposium, CNS '17, pages 9:1–9:12, San Diego, CA, USA, 2017. Society for Computer Simulation International.

27. Chen, T.: A factory simulation system based on cloud services and portable scheduling intelligence. In Proceedings of the 2016 8th International Conference on Information Management and Engineering, ICIME 2016, pages 85–88, New York, NY, USA, 2016. ACM.

28. Hwangbo, S. and Lee, K.: Cloud services for modeling and simulation: A simulation of a chemical gasdiffusion in the cloud. In Proceedings of the 20th International Symposium on Distributed Simulation and Real-Time Applications, DS-RT '16, pages 187–188, Piscataway, NJ, USA, 2016. IEEE Press.

29. Heavey, C., Dagkakis, G., Barlas, P., Papagiannopoulos, I., Robin, S., Mariani, M., and Perrin, J.: Development of an open-source discrete event simulation cloud enabled platform. In Proceedings of the 2014 Winter Simulation Conference, WSC '14, pages 2824–2835, Piscataway, NJ, USA, 2014. IEEE Press.

30. Helal, A. E., Bayoumi, A. M., and Hanafy, Y. Y.: Parallel circuit simulation using the direct method on a heterogeneous cloud. In Proceedings of the 52Nd Annual Design Automation Conference, DAC '15, pages 186:1–186:6, New York, NY, USA, 2015. ACM.

31. Tanenbaum, A. and van Steen, M.: Distributed Systems, Principles and Paradigms. 2nd. Prentice Hall, 2007. steen2007.00 Translations: German, Portugese, Italian.

32. Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM, 21(7):558–565, 1978.

33. Kshemkalyani, A. D. and Singhal, M.: Distributed Computing. Cambridge University Press, 2011.

34. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. 21(7):558–565, July 1978.

35. Pelosi, M.: Cast: Cloud-based applications simulator testbed. 2021.

36. Press., O. U.: Oxford Living Dictionary. Latency., 2017.

## CITED LITERATURE (continued)

37. Distributed systems: Principles and paradigms.

38. Mansouri, Y.: Brokering Algorithms for Data Replication and Migration Across Cloud-based Data Stores. (March), 2017.

39. Petersen, K., Spreitzer, M. J., Terry, D. B., Theimer, M. M., and Demers, A. J.: Flexible Update Propagation for Weakly Consistent Replication. Operating Systems Review (ACM), 31(5):288–301, 1997.

40. Elnozahy, M.: Time Synchronization and Logical Clocks.

41. Cristian, F.: Understanding fault-tolerant distributed systems. Commun. ACM, 34(2):56–78, February 1991.

42. V., H. and S., T.: Fault-tolerant broadcasts and related problems. In Distributed Systems (2nd edition), ed. S. Mullender. Addison-Wesley, 1993. pages 97–145.

43. Costan, A., Dobre, C., Pop, F., Leordeanu, C., and Cristea, V.: A fault tolerance approach for distributed systems using monitoring based replication. pages 451 – 458, 09 2010.

44. Lianza, T. and Snook, C.: A byzantine failure in the real world. `https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world`, 2020 (accessed January 22, 2020).

45. Nita, M. C., Pop, F., Mocanu, M., and Cristea, V.: FIM-SIM: Fault injection module for CloudSim based on statistical distributions. Journal of Telecommunications and Information Technology, 2014(4):14–23, 2014.

46. Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A. F., and Buyya, R.: Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. Softw. Pract. Exper., 41(1):23–50, January 2011.

47. Hazelcast actor. Available at GitHub: `https://github.com/truemped/hazelcast-actor` (2020/02/01).

48. Build powerful reactive, concurrent, and distributed applications more easily. Available at the Akka website: `https://akka.io` (2020/01/17).

## CITED LITERATURE (continued)

49. herd, a memory model simulator. Available at the Herd website: `http://diy.inria.fr/herd/` (2020/02/01).

50. Kathiravelu, P. and Veiga, L.: An Elastic Middleware Platform for Concurrent and Distributed Cloud and Map-Reduce Simulation-as-a-Service. page 12, 2014.

51. Kathiravelu, P.: An Elastic Middleware Platform for Concurrent and Distributed Cloud and MapReduce Simulations. (September), 2016.

52. Cloud simulation under fault constraints. Proceedings - 2014 IEEE 10th International Conference on Intelligent Computer Communication and Processing, ICCP 2014, pages 341–348, 2014.

53. Baragiola, A.: CAST (Cloud-based Applications Simulator Testbed): A detailed view ondomain-specific features, 2021.

54. Mell, P. and Grance, T.: The NIST Definition of Cloud Computing., 2009.

55. Nam, D.: Api design implications of boilerplate client code. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 1253–1255, 2019.

56. Chinosi, M. and Trombetta, A.: Bpmn: An introduction to the standard. Computer Standards and Interfaces, 34(1):124 – 134, 2012.

57. Hewitt, C., Bishop, P., and Steiger, R.: A universal modular ACTOR formalism for artificial intelligence. In Proc. International Joint Conference on Artificial Intelligence, pages 235–245, 1973.

58. Hewitt, C.: Actor Model of Computation: Scalable Robust Information Systems. 2010.

59. Agha, G.: An overview of actor languages. Proceedings of the 1986 SIGPLAN Workshop on Object-Oriented Programming, OOPWORK 1986, (October):58–67, 1986.

60. Spray: Elegant, high-performance http for your akka actors.

61. Build massively scalable soft real-time systems. Available at the Erlang website: `https://www.erlang.org/` (2020/01/17).

## CITED LITERATURE (continued)

62. Lea, D.: A Java fork/join framework. ACM 2000 Java Grande Conference, pages 36–43, 2000.

63. Scalability of fork join pool. Available at: `https://letitcrash.com/post/17607272336/scalability-of-fork-join-pool` (2020/11/22).

64. Grechanik, M.: Cloud Computing. 2019.

65. Baldoni, R. and Bonomi, S.: Distributed system logical time. Available at: `http://www.dis.uniroma1.it/~baldoni/Logical_Time.pdf` (2015/01/12).

66. Russ Miles, K. H.: Learning UML 2.0. O'Reilly Media, Inc., 2006.

67. Paul Bratley, Bennet L. Fox, L. E. S.: A Guide to Simulation. Springer Science Business Media, 2011.

68. Unified modeling language 2.5.1, 2017.

69. Kemme, B.: Data Replication, pages 626–630. Boston, MA, Springer US, 2009.

70. eds. M. T. Özsu and P. Valduriez Principles of Distributed Database Systems. New York, Springer, 3 edition, 2011.

71. Yu, H. and Vahdat, A.: Design and evaluation of a conit-based continuous consistency model for replicated services. ACM Trans. Comput. Syst., 20(3):239–282, August 2002.

# VITA

| | |
|---|---|
| NAME | Amedeo Baragiola |
| EDUCATION | |
| | Master of Science in Computer Science, University of Illinois at Chicago, May 2021, USA |
| | Master of Science in Computer Science and Engineering, May 2021, Polytechnic of Milan, Italy |
| | Bachelor of Science (Hons) in Engineering of Computing Systems, Jul 2018, Polytechnic of Milan, Italy |
| LANGUAGE SKILLS | |
| Italian | Native speaker |
| English | Full proficiency |
| | 2018 - IELTS examination (8.0/9.0) |
| | A.Y. 2019/20 One Year of study abroad in Chicago, Illinois |
| | A.Y. 2018/19. Lessons and exams attended exclusively in English |
| SCHOLARSHIPS | |
| Fall 2020 | Research Assistantship (RA) position (20 hours/week) with full tuition waiver plus monthly stipend |
| Spring 2020 | Research Assistantship (RA) position (20 hours/week) with full tuition waiver plus monthly stipend |
| Fall 2019 | GA hourly position at UIC Innovation Center (10-15 hours/week) with bi-weekly stipend |
| TECHNICAL SKILLS | |
| Languages | C/C++, JavaScript, Scala, Java, Perl, Python, SQL. |
| Frameworks | Akka, Apache Spark, Apache Hadoop, Flask, Spring, Android, node.js. |
| Developer Tools | Google Cloud Platform, AWS, Microsoft Azure, Cloudera HDP Sandbox, Git, Docker, TravisCI, IntelliJ, Android Studio, Eclipse. |