Large Scale Distributed Algorithms Simulator

Inês Amaral Sequeira

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Miguel Ângelo Marques de Matos

November 2019

For my parents,

Acknowledgments

I would like to thank my advisor Professor Miguel Matos for all the guidance throughout this work. I would also like to thank my friends. Last but not least, I would like to thank my parents, my sister and my brother. I could not have made it without their support.

This work is partially funded by Fundo Europeu de Desenvolvimento Regional (FEDER) via Programa Operacional Regional de Lisboa and by Fundos Nacionais via FCT - Fundação para a Ciência e a Tecnologia through project Lisboa-01-0145-FEDER-031456 (Angainor) and UID/CEC/50021/2019.

Resumo

Os sistemas distribuídos, e os algoritmos distribuídos subjacentes, estão na base de inúmeros serviços usados hoje em dia, como Computação em Nuvem, Redes Sociais ou Cripto-moedas. Devido à sua importância, é fundamental que os algoritmos distribuídos sejam devidamente avaliados para garantir que funcionam como especificado, mesmo na presença de condições adversas. A simulação é uma abordagem útil para fazer essa avaliação, em particular nas fases iniciais de desenvolvimento, pois permite testar os algoritmos num ambiente completamente controlado. Infelizmente, os simuladores existentes têm várias lacunas como, por exemplo, modelos de simulação irrealistas, pouca escalabilidade ou falta de funcionalidades fundamentais como a modelação de assincronia entre processos.

Nesta tese, propomos o Corten, um novo simulador, desenhado de raiz, que preenche as lacunas acima identificadas, produz simulações reproduzíveis e é eficiente, permitindo simular milhares de nós numa máquina com recursos modestos. Na nossa avaliação ilustramos as principais funcionalidades do simulador e demonstramos a sua aplicação por comparação ao estado da arte.

Palavras-chave: Algoritmos Distribuídos, Simulação, Larga Escala, Avaliação

Abstract

Distributed systems, and the underlying distributed algorithms, are at the base of countless services used today, such as Cloud Computing, Social Networks or Criptocurrencies. Due to their importance, it is fundamental that distributed algorithms are evaluated properly, to ensure that they work as specified, even in the presence of adverse conditions. Simulation is a useful approach to do that evaluation, particularly in the initial phases of development, because it allows for the algorithms to be tested in a completely controlled environment. Unfortunately, existing simulators have several shortcomings, for example, unrealistic simulation models, poor scalability or lack of fundamental functionalities, such as modelling asynchrony between processes.

In this thesis, we propose Corten, a new simulator, designed from scratch, which overcomes the shortcomings mentioned above, produces reproducible simulations and is efficient, allowing to simulate thousands of nodes in a machine with modest resources. In our evaluation, we illustrate the main functionalities of the simulator and demonstrate its use in comparison with the state of the art.

Keywords: Distributed Algorithms, Simulation, Large Scale, Evaluation

Contents

Acknowledgments . . . v

Resumo . . . vii

Abstract . . . ix

List of Tables . . . xiii

List of Figures . . . xv

Glossary . . . xvii

1 Introduction 1

2 Related Work 3

2.1 Concepts . . . 3

2.2 Simulation only . . . 4

2.2.1 PeerSim . . . 4

2.2.2 Optimal-sim . . . 5

2.2.3 PlanetSim . . . 5

2.2.4 NDP2PSim . . . 6

2.2.5 D-P2P-Sim . . . 7

2.2.6 PeerfactSim.KOM . . . 8

2.2.7 DistAlgo . . . 8

2.3 Simulation and real execution . . . 9

2.3.1 ProtoPeer . . . 9

2.3.2 Neko . . . 10

2.3.3 RealPeer . . . 10

2.4 Emulation and real execution . . . 11

2.4.1 OverlayWeaver . . . 11

2.5 Real execution . . . 12

2.5.1 SPLAY . . . 12

2.5.2 P2 . . . 13

2.5.3 Mace . . . 13
2.6 Summary . . . 14

3 Corten **17**
3.1 Architecture . . . 17
3.2 Random Number Generation . . . 22
3.3 Event Queue Structure . . . 22
3.4 Interactivity and Extensions . . . 23
3.5 Discussion . . . 23

4 Evaluation **25**
4.1 Micro-benchmarks . . . 25
4.2 Macro-benchmarks . . . 29
4.2.1 Case study: Chord . . . 29
4.2.2 Comparison with PeerSim . . . 30

5 Conclusions **33**
5.1 Future Work . . . 33

Bibliography **35**

A Echo Application **39**

List of Tables

2.1 Overview of the state-of-the-art tools 15

List of Figures

2.1 Common API of PlanetSim 6

3.1 Message sending in Corten 18
3.2 Method call in Corten 19
3.3 Corten's Architecture 19

4.1 Churn configuration used in the churn experiment 26
4.2 Configuration used in the churn experiment 26
4.3 Churn micro-benchmark 27
4.4 Churn from the perspective of a single process 27
4.5 Network Latency micro-benchmark 28
4.6 Process Asynchrony micro-benchmark 29
4.7 Memory usage for Corten and PeerSim 31
4.8 Real time duration of Corten and PeerSim simulations 32

Glossary

API	Application Programming Interface
DHT	Distributed Hash Table
DOLR	Decentralised Object Location and Routing
GUI	Graphical User Interface
KBR	Key-based Routing API
P2P	Peer-to-Peer
RNG	Random Number Generator

Chapter 1

Introduction

Distributed applications and systems have an increasingly important role in the development of modern computational systems. Distributed systems are ubiquitous, continuously growing and becoming more complex. Therefore, it is important to build robust and dependable distributed systems, because distributed systems are at the centre of several areas, such as the technological and financial areas. Due to the importance of distributed systems, and consequently the importance of the distributed algorithms on which they rely, it is necessary to guarantee that the algorithms behave correctly under the specified models. However, the inherent difficulty in designing correct algorithms led to the emergence of several techniques and tools to help the designer.

A distributed algorithm can be described through a high-level specification language, such as TLA$^+$ [LMTY02], and it can be formally validated through the use of tools, such as TLC [LMTY02]. However, there is a gap between this specification and an implementation of the same algorithm, due to simplifications in the specification and due to the fact that there is not a direct translation between a specification and an implementation. The same can be said for the relation between the pseudocode of an algorithm presented in a scientific paper and the implementation of the same algorithm. A good specification does not necessarily imply that the implementation does what it is supposed to do according to the specification. Moreover, a specification does not always model, or allow to model, what happens in adverse situations, such as churn or packet loss. Due to this fact, there is a need to test and evaluate distributed algorithms in controlled environments.

Tools such as simulation can be used to facilitate the development and testing of distributed algorithms. In simulation, there is total control over the execution environment, for example, it is possible to control churn and submit the algorithm to several adverse situations that can be found in practice. There is a vast number of simulation tools with different goals and models. However, current simulators have several limitations: poor scalability, lack of fundamental functionality, such as modelling asynchrony between processes, or simplifying assumptions that can lead to incorrect designs if not used with care. For example, PeerSim [MJ09] allows for applications to make calls to methods of other nodes directly, therefore modelling atomic message exchanges which is unrealistic in most models.

In this thesis, we solve these problems with Corten, a new simulator which covers the following functionalities: models several types of churn; models network asynchrony (latency, jitter, packet loss); models process asynchrony, which, to the best of our knowledge, no other simulator currently supports; checkpointing, allowing to save a snapshot of the simulation and restore it at a later point in time. Each experience in the simulator is reproducible, making it possible to, for example, repeat an experiment in which there was an unexpected result and study it in more detail. The reproducibility characteristic makes it possible for third parties to verify the obtained results. Corten is available as open-source in `https://github.com/miguelammatos/corten`.

The rest of this document is organised as follows. Chapter 2 presents a revision of related work. Chapter 3 describes Corten's architecture and implementation. Chapter 4 defines the algorithms and metrics used for Corten's evaluation and describes the obtained results. Chapter 5 presents the conclusions and future work.

Chapter 2

Related Work

There are several ways to evaluate and develop distributed algorithms, namely using simulators, emulators and real executions. We divided the discussion into sections in order to focus on each type of solution at a time and to have a clear way of specifying which category the systems discussed belong to: *Simulation only*, *Simulation and real execution*, *Emulation and real execution*, and *Real execution*. We introduce core concepts in Section 2.1 and discuss the above mentioned approaches in Section 2.2 to Section 2.5.

2.1 Concepts

In this section, we introduce some core concepts that will be used throughout the rest of the document.

There are several alternatives for testing and evaluating a distributed system: simulation, emulation and executing a real system in a testbed. Simulation is the process of designing a model of a real system and conducting experiments with this model for the purpose of understanding the behaviour of the system and/or evaluating various strategies for the operation of the system [KHZ$^+$05]. A network topology is the arrangement of elements, such as nodes and links, of a communication network. Simulation does not directly represent the underlying network topology for data transmission between peers, but tries to account for influencing factors such as latency and jitter. A simulator abstracts the network through a simplified interface. Message passing calls are passed to the simulator kernel and added to a queue. There are no actual packets being sent and received. An emulator provides a fully fledged network interface and typically provides its own implementation of the network stacks. Emulators are more complex to implement but allow for a more detailed representation of the system and capture the structure of the network more closely. Simulators can be used to test more complex systems since they avoid the details of the network implementation by replacing it with a queue based model. For that same reason they are more scalable than emulators. A testbed is a controlled environment for testing systems. The execution of experiments in testbeds suffers scalability, reproducibility and requires existing prototypes [SGR$^+$11].

In the context of Peer-to-Peer (P2P) applications, it is common for nodes to join and leave the network frequently. This is called churn. Moreover, P2P applications are typically built on top of an overlay.

An overlay network is a computer network on top of another network. The nodes in an overlay network are connected with one another through logical or virtual links. Each of these links correspond to a path in the underlying network.

2.2 Simulation only

In this section, we discuss systems that provide a means to develop applications for simulation, and where the code for simulation cannot be used directly to run in a real setting. We use the term simulation only to state that the code for simulation without any modification cannot be run in a real environment.

2.2.1 PeerSim

PeerSim [MJ09] is a simulator for P2P systems. PeerSim models a network as a list of nodes. In PeerSim, the actions each node will perform are called protocols. Protocols are defined by the developers, by implementing a specific Java interface. In PeerSim, a simulation contains initialisers and controls. Initialisers are executed before the simulation and controls are executed during the simulation. They can be used to monitor or modify components. PeerSim provides cycle-based and event-based engines. In the cycle-based model, protocols from the participating nodes are executed in a round-robin fashion. In the event-based model, events or messages are executed according to a specified chronological order. Simulations are described by plain text configuration files. PeerSim can use trace-based datasets, in order to reproduce an experiment. It can simulate failures and churn. It supports graph representations of the overlay networks, which can be exported for visualisation.

On the one hand, the cycle-based engine scales well but it is not very realistic, on the other hand the event-based engine is more realistic but does not scale as much. The cycle-based engine is not very realistic because it does not simulate the transport layer and does not provide concurrency, i.e. nodes communicate directly with one another and, in some sequential order, each node performs some actions, such as calling methods and performing computations. In the event-based engine, PeerSim has to handle message sending and each control has to be scheduled explicitly.

2.2.2 Optimal-sim

Optimal-sim [WI05] is used to generate overlay network topologies and simulate behaviours of nodes. Optimal-sim is implemented in Java. The underlying network topologies are generated by a topology generation tool, called BRITE [MM00]. Optimal-sim is able to simulate a network with up to 10 000 nodes. Optimal-sim is an event-based simulator and can simulate the join, departure and failure of nodes. A simulation can be described in a simulation scenario file, which contains the events to be executed. A description of a P2P algorithm can be written using the API provided by Optimal-sim. P2P overlay networks built by peer nodes at the application level are generated by Optimal-sim. The authors define the following metric: average hop count, which is an important indicator to evaluate efficiency, and number of messages transferred. A topology optimisation mechanism, developed by the authors, was tested using Optimal-sim. With this optimisation mechanism, a peer node reconfigures the topology based in the neighbour nodes' information. Both the virtual links at the P2P level and the hop count of physical links between peers are considered in the mechanism. This mechanism has some awareness of the underlying network topology and by using it a reduction in the average hop count between any pair of peer nodes was achieved. This indicates that Optimal-sim is useful and can be used to test and compare different overlay algorithms and optimisations to existing algorithms.

2.2.3 PlanetSim

PlanetSim [GPM$^+$05] is a simulation framework for overlay networks and services. PlanetSim was developed in Java. PlanetSim is divided into Application Layer, Overlay Layer and Network Layer. By using layers and the Common API [DZD$^+$03], PlanetSim is able to decouple services built in the application layer and P2P algorithms in the overlay layer. The Common API provides a layer of abstraction between the services and the overlay algorithms. The Common API tries to identify the fundamental abstractions provided by structured overlays and to define APIs for the common services they provide. Within the Common API there is the Key-based Routing API (KBR). The KBR represents the basic capabilities which are common to all structured overlays. The goal of the Common API is for all overlay protocols to implement the KBR. By having all overlay protocols implement the KBR, they can be used interchangeably, i.e. applications that are built on top of overlays can use and be experimented on top of any overlay. For example, in Figure 2.1 we have Application A which uses Overlay A through KBR calls, if we change

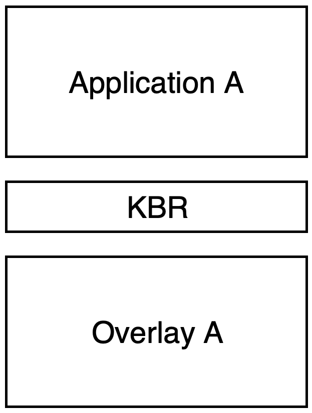

Figure 2.1: Common API of PlanetSim

Overlay A to Overlay B, Application A still works, as long as both Overlay A and Overlay B implement the KBR. PlanetSim uses FreePastry's version of the Common API. FreePastry is an open-source implementation of Pastry [RD01] intended for deployment in the Internet. FreePastry is intended primarily as a tool that allows interested parties to evaluate Pastry and as a platform for the development of applications. Since PlanetSim uses FreePastry's version of the Common API, services and applications made for FreePastry can be used in PlanetSim and experimented on overlay algorithms other than Pastry.

In each step of the simulation, outgoing messages are moved to incoming queues for all nodes. The authors state that, to change from simulation to a real system, the network implementation could be changed from a simulated network into a network which routes the messages using appropriate TCP or UDP connections on top of a real IP network, but this was not implemented. PlanetSim is able to simulate churn and can serialise the full state of a simulation to a file. This is useful for large simulations and saves computing time, since it can be used after stabilising a huge overlay and later on resume the simulation from that point. PlanetSim has Chord [SMLN+03] and Symphony [MBR03] overlays implemented, and services such as CAST, Distributed Hash Table (DHT) and Decentralised Object Location and Routing (DOLR).

PlanetSim separates in a clear way the implementation of overlay algorithms and the implementation of services on top of existing overlays. The idea of saving a snapshot of the system at any given time, and then being able to resume the execution from that point on is very interesting and is incorporated in our simulator.

2.2.4 NDP2PSim

NDP2PSim [KHZ+05] is an integrated simulation platform used to develop P2P applications, which besides including an application layer also offers support for working with a detailed underlying network model. NDP2PSim uses Tcl/Otcl as its user interface language. NDP2PSim builds on top of NS-2 [FV01], a generic network simulator, and adds application layer modules. NS-2 is used to specify the underlying network. In the underlying network, there are application peers and routing peers. Routing peers forward the messages. Application peers have application components. Generating the underlying network topology to be used for a particular simulation can be facilitated by the use of a topology-producing tool, GT-ITM [CDZ97].

NDP2PSim comprises Socket Module, Interfaces Module, P2P Application Module, Message Handler, and Statistics Module. Socket Module simulates socket actions, supports message transmission across peers and congestion control. Interfaces Module encapsulates the sending and receiving of messages, doing the translation from application to sockets and vice versa. P2P Application Module is divided into different functions P2P applications can have: assigning files to peers according to a certain distribution model; establishment and destruction of logical links; file store and search functions, and application multicast functions. P2P Application Module also provides an out-of-band mechanism for peers to communicate, for example for choosing peers' neighbours before the peers join the network. Message Handler is used to create and parse application-layer messages. The Statistics Module collects various kinds of data, such as the number of control messages and data messages, of each peer, which are saved in files. The user can add new data to be collected if necessary. NDP2PSim offers support for visualising network simulation traces, using NAM (Network AniMator) [FV01].

Since NDP2PSim is built on top of NS-2, it can only scale as much as NS-2, that is, up to about 5 000 nodes. This is due to the NS-2 detailed underlying network model that sacrifices some of the efficiency for detail, resulting in poor scalability for NDP2PSim.

2.2.5 D-P2P-Sim

D-P2P-Sim [SPS+09] is a distributed simulation environment for P2P simulations. D-P2P-Sim is implemented in Java. D-P2P-Sim is event-driven and uses a pool of threads in order to minimise thread creation overhead and thread competition. Every time a message is sent, a thread is assigned to the message's destination peer to serve that particular message. D-P2P-Sim can be used in a single machine or in several machines. There is a 10% overhead when using multiple machines due to the coordination and communication of processes. D-P2P-Sim offers a GUI for choosing the parameters of the simulation and to control the execution. D-P2P-Sim generates statistical charts automatically after a simulation. D-P2P-Sim provides an extensible API, which is based on the plugin mechanism of Java.

Over 400 000 nodes can be simulated in a single machine, but D-P2P-Sim does not take packet loss and latency into consideration. The GUI is only available when the simulation occurs in a single machine.

2.2.6 PeerfactSim.KOM

PeerfactSim.KOM [SGR$^+$11] is a Java-based simulator for investigating large-scale P2P systems. PeerfactSim.KOM is an event-based simulator. It has a layered architecture, providing interfaces between the layers. PeerfactSim.KOM provides default implementations of the layers when they can be reused without any change in the simulation. When this is not possible, PeerfactSim.KOM provides a base implementation that needs to be extended to achieve the desired functionality. PeerfactSim.KOM separates Overlay, Services and Application into different layers so that the user can focus on each separately or focus in just one while using available implementations for the others. After the execution of a simulation, it is possible to visualise the topology and the exchanged messages from that simulation.

2.2.7 DistAlgo

DistAlgo [LSL17] is a language for the development of distributed systems. DistAlgo extends Python with high-level features for describing distributed algorithms. The compiler applies analyses and optimisations, especially to the high-level queries, and generates executable Python code. DistAlgo supports high-level control flows where complex synchronisation conditions can be expressed using high-level queries, namely logic quantifications, over message history sequences. Distributed algorithms can be expressed in DistAlgo clearly at a high level. They can also be formally verified, since the language possesses a clear operational semantics. DistAlgo combines the advantages of pseudocode, formal specification languages, and programming languages. Yield points are places in the DistAlgo code where received messages can be handled. Yield points can be specified explicitly and declaratively. During compilation, appropriate message handlers are inserted at each yield point. Synchronisation conditions are expressed by the use of high-level queries: quantifications, comprehensions and aggregations. In order to optimise quantifications they are converted into aggregations and then optimised. Incrementalisation transforms expensive computations into efficient incremental computations with respect to updates to the values on which the computations depend. DistAlgo maintains a message history containing all messages sent and received by a process. This has the potential to grow unboundedly, and it is handled by maintaining only auxiliary values needed for incremental computation. DistAlgo programs tend to be smaller than programs written in other languages. The authors claim that in some cases DistAlgo programs are a half to a fifth of the size of comparable programs in other languages.

Quantifications are dominantly used in writing synchronisation conditions and assertions in specifications, but they are not used in programming languages because they are not easy to implement efficiently. DistAlgo solves this by providing quantifications, which the compiler transforms into aggregations, and then they can be incrementalised in order to be more efficient. In terms of execution time and considering the tested algorithms, DistAlgo programs run considerably slower, taking more than twice as long as the equivalent in other programming languages.

The language of DistAlgo is concise yet expressive. As a future development of Corten we are considering providing a high-level language with similar characteristics to DistAlgo in order to facilitate the programming of Corten applications.

2.3 Simulation and real execution

In this section, we discuss systems that provide a means to develop applications for simulation, in such a way that these applications can also be executed in a real setting without any code modifications.

2.3.1 ProtoPeer

ProtoPeer [GADK09] is a simulator implemented in Java used for prototyping, building and evaluating P2P systems. Message passing logic and the state of each of the protocols is encapsulated in components called peerlets. A peer can be composed of several peerlets, and each peerlet can be reused across applications. ProtoPeer provides APIs for message passing, message queuing, timer operations, overlay routing and managing the overlay neighbours. Both time and network are abstracted through APIs. Switching from the simulated system to the actual system can be done by switching from one time and networking implementation to another. ProtoPeer supports packet loss and delay. Developers can personalise the implementation of packet loss and delay. ProtoPeer can simulate churn and failures. To do this, ProtoPeer uses events. Each event is specified by a triple which consists of time, the set of unique peer indices to be affected, and the method to call. A set of events defines a scenario, which is loaded on startup by the peers. Peers execute the events in the scenario. ProtoPeer provides a measurement API, that allows to obtain measurements, in order to evaluate the system. ProtoPeer is complex and memory-intensive, limiting scalability.

In ProtoPeer, no code needs to be changed in order to switch from simulation of a P2P system to its deployment in a real network.

2.3.2 Neko

Neko [UDS01] is a communication platform for prototyping and simulating distributed algorithms. Neko is implemented in Java. In Neko, there are two main parts: application and networks. There are several processes at the application level. Each process has layers. Messages are used for the communication between layers within processes and between different processes. For each process, there is a Process object that connects it to the network. This object is responsible for holding some process wide information, such as the address of the process, for implementing message logging services, and dispatching and collecting messages from the network. Configuration files describe all the aspects of an application, including the number of processes and the network protocol to be used. In a real implementation, Java Threads could be used, whereas in a simulation one would have to use a simulation package, but these details are abstracted away by Neko's API for threads. Thus, by using Neko's API for threads, an application can be used in simulation and deployed in a real network without any code changes. Neko was deliberately kept simple, easy to use and extensible, and since it is written in Java, it is highly portable.

The same code, without any change, can be used for both simulation and execution on a real network. Neko is focused more on the application part, rather than on the network. The network was not designed for efficiency but for simplicity. In a simulation, all processes run on a single JVM, hence simulations are not distributed, they only simulate distribution. Therefore, there is no isolation between the processes during simulation, and because of this developers must be careful when using static variables.

2.3.3 RealPeer

RealPeer [HBH07] provides a means to develop P2P systems from simulation to a real system. RealPeer is implemented in Java. For RealPeer, the central element is the P2P application, which is represented in such a way that it can both be used as a model in simulation and a real application in a real system. A P2P application in RealPeer is divided into four layers: P2P Core, which has responsibilities such as maintaining the overlay network and routing; P2P Services, which is responsible for common services such as resource management; Application, which has the specific functionality within the application domain, and User Interface, which provides an interface to the user. In the beginning of development, each layer is a very simple version of the system. In RealPeer, an initial model of a P2P system is iteratively transformed into the intended real P2P system. Different layers can be in different stages of development. This gives the opportunity to test a real application with a model of a network, for example, and many other combinations of parts of a real system and parts of a model. Elements of the domain model are represented by hot-spots. The developer can extend the hot-spots by implementing plug-ins for them. A central scheduler is used to control system time and execution of the model. By the way it is designed it does not allow for concurrent execution. According to the authors, memory consumption seems to be the major bottleneck when performing experiments.

2.4 Emulation and real execution

In this section, we discuss systems that provide a means to develop applications for emulation. Applications developed for emulation can also be executed in a real setting.

2.4.1 OverlayWeaver

Overlay Weaver [STS08] was created with the goal of facilitating the rapid development of realistic overlay algorithms and their applications. Overlay Weaver is implemented in Java. The routing process was separated from the routing layer and a programming interface was created between the common routing process and routing algorithms. Overlay Weaver provides implementations of the common routing process, so that the developers can use their API and focus their coding efforts on the algorithms themselves. OverlayWeaver provides both an iterative and a recursive implementation of the common routing process. The routing driver is a component that conducts the common routing process. In Overlay Weaver, the operations to be performed are described in an emulation scenario. A scenario can be produced by the user or generated automatically. The emulator can be run on a single machine or on multiple machines. In both cases, the emulator reads and executes the same emulation scenario. Overlay Weaver provides a distributed environment emulator, which can host thousands of nodes on a single computer. Developers can improve new algorithms and their implementations rapidly by testing them iteratively on the emulator. Overlay Weaver provides a Messaging visualiser, which can be used to show nodes and communication among them. Overlay Weaver provides implementations of well known routing algorithms Chord [SMLN$^+$03], Pastry [RD01], Tapestry [ZHS$^+$04], and Kademlia [MM02]. The authors have implemented these algorithms with just a few hundreds of lines of code each.

The visualiser provided by Overlay Weaver imposes a burden on the emulator by visualisation in addition to doubling the number of messages. This results in reducing the maximum number of emulated nodes [SJG17].

2.5 Real execution

In this section, we discuss systems that provide a means to develop applications that can execute in a real setting. Although the main focus of our work is simulation, in the end the purpose of any distributed application is to be executed in a real setting. Our proposed simulator, Corten, is designed in such a way that it would be possible to exchange the simulated network with a real one, even though, due to time constraints, the actual translation to a real network will not be implemented.

2.5.1 SPLAY

SPLAY [LRF09] simplifies the prototyping, development, deployment and evaluation of distributed algorithms. SPLAY applications are written in a lightweight platform-independent language called Lua. SPLAY provides libraries for the development of distributed protocols. In order to prevent SPLAY applications from accessing data or resources on the host they are not supposed to, SPLAY provides its own versions of some libraries in order to offer restricted access to system resources, like files, in an OS-independent way. SPLAY applications are sandboxed. SPLAY allows for deployments where some nodes may be part of different testbeds. SPLAY's infrastructure manages to hide away much of the complexity, therefore greatly simplifying the implementation of SPLAY applications, which tend to be concise. SPLAY has three main components: controllers, daemons and applications. The controller manages the deployment and execution of applications. A daemon process runs on every machine of the testbed. When instructed by the controller, a daemon process instantiates, stops and monitors applications. Many SPLAY applications can run on the same host simultaneously. Controllers keep track of the active daemons and applications. There can be several controllers. Data for all the hosts and applications are stored in a database shared by all the controllers. Churn management provides a means to start and stop processes on demand by sending instructions to daemons. It is possible to reproduce an experiment, by using traces or scripts. SPLAY provides a log library that allows the developer to print information locally or to send it over the network to a log collector, in order to debug and collect statistics.

SPLAY applications' code looks similar to research papers' pseudocode. However, since SPLAY uses Lua and thread support in Lua is not preemptive, SPLAY has to rely on cooperative multitasking. Coroutines are supposed to yield the processor to one another when they are about to do an operation that might block. Thus, it is the responsibility of the programmer to write code that yields the system resources at appropriate places.

2.5.2 P2

P2 [LCH$^+$05] is a tool for developing systems that are based on overlay networks and will execute in a real setting. P2's goal is to explore the feasibility of the declarative approach in practice at a coarse grain, without worrying about all possible optimisations. P2 takes a concise logical description, written in a language designed by the authors, OverLog. Then P2 translates it into a dataflow graph, which maintains overlays at runtime. Since P2 specifications are of a logical nature, they can be decomposed into logically reusable units. P2 uses tuples and tables to store information about the network. P2 also provides logging and debugging mechanisms. According to the authors, P2 provides acceptable performance, compared to hand-coded implementations, despite the simplicity of the specification.

P2 can be used for prototyping systems and eventually for deploying them, but it does not provide support for deployment, i.e. the user needs to write scripts to deploy the system in a real setting [LRF09]. P2 specifications are succinct but their performance is not as good as other solutions [KAB$^+$07]. Since OverLog is a high-level language, it hides most of the low-level implementation details.

2.5.3 Mace

Mace [KAB$^+$07] is a language extension for C++ and source-to-source compiler, that supports building robust and high-performance distributed systems. It allows the compilation of readable high-level descriptions into implementations with performance similar to that of hand-coded implementations. Mace is designed around three main entities: objects, events and aspects. Service objects are connected through interfaces. Each event corresponds to a method implemented by a service object. Aspects define tasks that need to be performed when certain conditions are satisfied. Aspects are used to separate code for debugging and log statistics, for example, from the event handling implementation. Service objects correspond to layers. Each service object is specified as a state transition system, where transitions represent the execution of methods corresponding to received events. Systems' specifications in Mace provide semantic information, which is used to generate code. Mace can also generate code for failure detection and handling. This reduces complexity and helps maintaining application consistency. Mace provides message serialisation and event dispatch, so developers do not have to implement this, which means the implementation size is reduced. Mace's layering mechanism and state-event semantics make it easier to use model checking to detect liveness bugs. The authors of Mace created a model checking tool, MaceMC [KAJV07], and used it to detect bugs in some systems.

In Mace, the system is decomposed into layers and interfaces, allowing for the reuse of subsystems implementations in different systems. The task of detecting, notifying and handling failures is simplified by the use of service objects. Mace provides good performance results, however it does not provide any built-in facility for deploying the system [LRF09].

2.6 Summary

In this section, we compare the systems discussed above and also our proposed simulator, Corten.

Table 2.1 summarises the tools based on characteristics that they possess, where each row corresponds to a tool and each column corresponds to a characteristic. For a certain tool, in its row, if in a given column there is a ✓then it means that that system has the characteristic from that column. If there is a ?, then we were not able to find sufficient information to know if the system has that characteristic or not. If it is left blank, then the system does not have that characteristic. The column "simulation" means that that tool provides a simulator. The column "emulation" means that that tool provides an emulator. The column "real execution" means that an application made for that tool can be executed in a real setting. The column "packet loss" means that packet loss is simulated or emulated by the tool. The column "latency" means that latency is simulated or emulated by the tool. The column "jitter" means that jitter is simulated or emulated by the tool. The column "reproducibility" means that multiple executions of an application with the same initial parameters have the same outcome. The column "number of nodes" indicates the maximum number of nodes achieved for the tool. The column "churn" indicates whether the tool provides churn capabilities. The column "logging" indicates if logging is available in the tool. The column "asynchrony" indicates if the tool takes into consideration that a computation can take an arbitrary time to finish. The column "checkpointing" indicates if the tool has the possibility of saving and loading the state of the application.

In summary, most tools implemented in Java do not scale better due to Java's memory overhead, therefore fewer nodes can be simulated or emulated in a single machine.

None of the discussed systems take asynchrony into consideration. As far as we know, asynchrony is a novel feature that we are taking into consideration in our simulator. Only one of the discussed systems, PlanetSim, can save the state and load it multiple times to test different scenarios from a common start point. This is important to reduce the time each subsequent simulation will take since the common part of each simulation can be done just once, the state can be saved, then that state can be loaded multiple times and different experiments can be done over that state.

Our proposed simulator, Corten, is the only one that covers all the characteristics mentioned in the Table 2.1, apart from emulation.

Table 2.1: Overview of the state-of-the-art tools

	simulation	emulation	real execution	packet loss	latency	jitter	reproducibility	number of nodes	churn	logging	asynchrony	checkpointing
PeerSim	✓			✓	✓	✓	✓	10^6 [*] 2.5×10^5 [**]	✓	✓		
Optimal-sim	✓			?	?	?	?	10 000	✓	✓		
PlanetSim	✓						✓	100 000	✓	✓		✓
NDP2PSim	✓			✓	✓	?	?	480	✓	✓		
D-P2P-Sim	✓						✓	400 000[†]	✓	✓		
PeerfactSim.KOM	✓			✓	✓	✓	✓	50 000[‡]	✓	✓		
DistAlgo	✓			?	?	?	?	?	?	✓		
ProtoPeer	✓		✓	✓	✓	✓	✓	50 000	✓	✓		
Neko	✓		✓	?	?	?	?	[§]	?	✓		
RealPeer	✓		✓				✓	20 000	?	✓		
OverlayWeaver		✓	✓	✓	✓	✓		4 000	✓	✓		
SPLAY		[¶]	✓				✓	500[†]	✓	✓		
P2			✓							✓		
Mace			✓							✓		
Corten	✓		✓[‖]	✓	✓	✓	✓	10^6	✓	✓	✓	✓

Our goal with Corten is to support all these features while being able to scale to a very large number of nodes. In the next section, we present how this is achieved.

[*] cycle-based
[**] event-based
[†] per host
[‡] according to [SGR+11]; 10^6 peers for simple overlays 10^5 peers for complex overlays, according to [SJG17]
[§] information not available
[¶] facilitates the integration with emulators
[‖] in a future version

Chapter 3

Corten

In this chapter, we describe Corten in detail. Section 3.1 describes Corten's architeture. Section 3.2 discusses the importance of Random Number Generation in simulation and the approach taken in Corten. Section 3.3 discusses the event structure used. In Section 3.4, we describe how the user can interact with Corten and how extensions can be implemented. We end this chapter with a discussion of the main functionalities of Corten in Section 3.5.

3.1 Architecture

In this section, we describe Corten's components in detail.

Corten is a simulator designed in a modular way. Each component is responsible for a specific functionality. This choice allows us to achieve good efficiency and flexibility. Corten is extensively configurable by the user to fit the needs of each particular simulation.

In Figure 3.1, there is an execution flow of a message being sent. For each application there is an associated process. (1) When an application wants to send a message it will call the `send` method of the corresponding process. (2) The process will then ask the network model for the latency of the message, which is how long the message will take to go from source to destination. (3) Then the process will add the event with the message to the event queue. (4) At a later point in time, the Simulation Kernel will get the next event to be executed from the event queue. It will give the event to the destination process by calling the `receive` method of the destination process. (5) The process will call the corresponding method in the application.

In Figure 3.2, there is an execution flow of a local method call. (1) When an application wants to make a local method call, it will call the method `call` of the corresponding process. (2) The process will ask the asynchrony model for the asynchrony of that call. The asynchrony model will calculate the asynchrony according to some distribution. (3) Then the process will

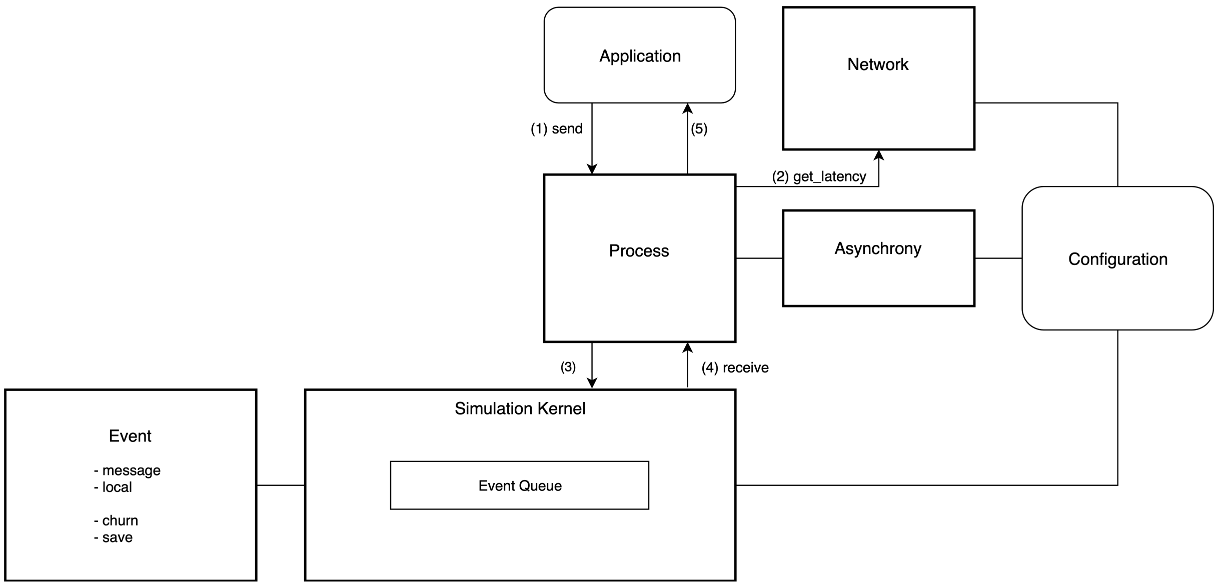

Figure 3.1: Message sending in Corten

add the event to the event queue. (4) At a later point in time, the Simulation Kernel will get the next event to be executed from the event queue. It will give the event to the destination process by calling the `receive` method of the destination process. (5) The process will call the corresponding method in the application.

Corten's components are represented in Figure 3.3, along with their main methods.

Application An application contains user defined code and interacts with the system through the interface exposed by Process.

An application needs to implement the following methods:

`init`: invoked when the application becomes active for the first time (i.e. after join).

`leave`: invoked when the application knows it is going to become inactive, this gives the application the possibility of warning other applications that it is going to become inactive in case it is relevant to the algorithm. Models a controlled exit from the system.

`recover`: invoked when an application had become inactive previously and it wants to become active again. Models *crash-recover* systems.

`on_load`: invoked when saved state from a previous simulation is loaded.

Process Each application is associated to a process. The process acts as a bridge between the application and the remaining components. Each process has a unique identifier.

The `send` method sends a message to a destination application identified by the process identifier. This message is the operation that will be executed in the destination. The `receive`

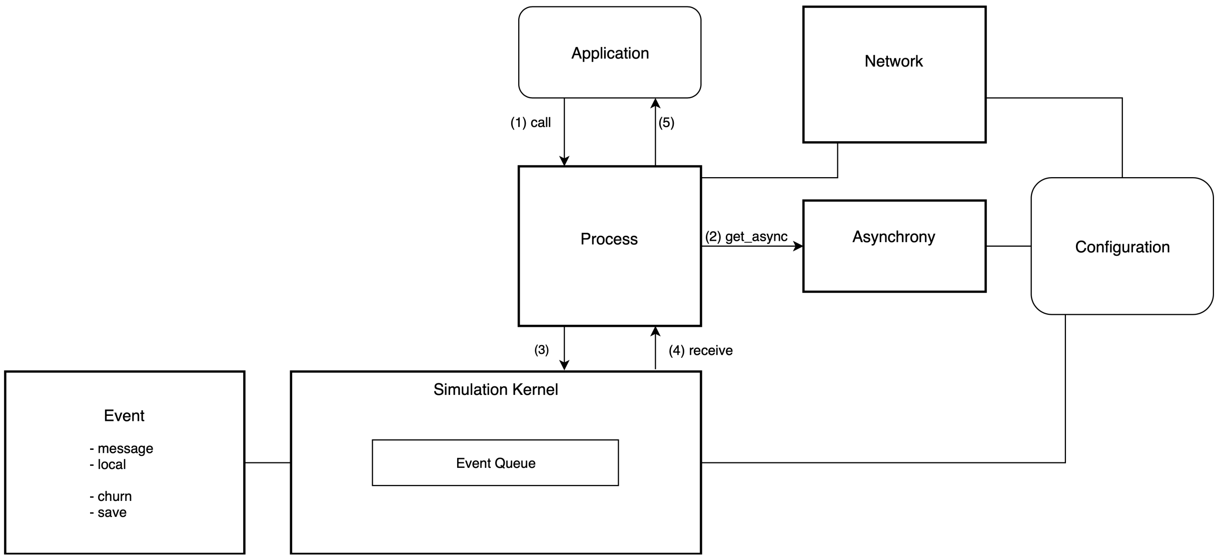

Figure 3.2: Method call in Corten

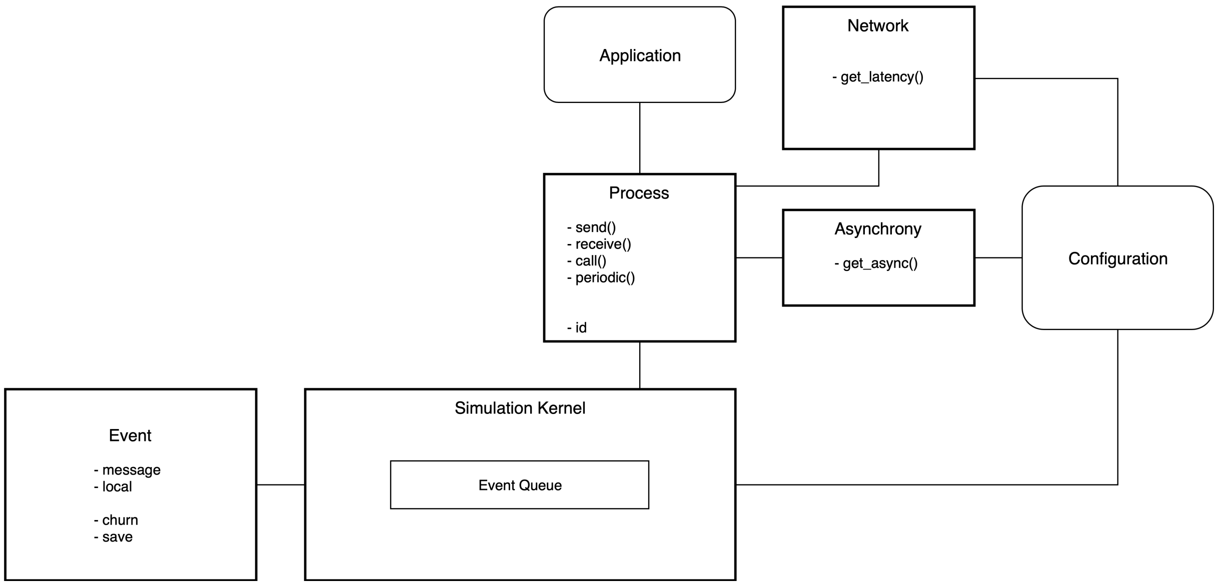

Figure 3.3: Corten's Architecture

method receives an event which contains the operation to be executed. This method has the role of calling the operation to execute in the target application. The `call` method and the `periodic` method are used for making local application calls. In both methods, it is specified the time after which the calls should be executed (*delta*), and for the `periodic` method it is specified how many times the method should be executed (*count*). A periodic method can be executed an infinite amount of times, in which case the *count* is zero. In this case, the simulation must be explicitly stopped, either from a save event which saves and stops (defined in configuration), or from an end event (defined in churn configuration). The above mentioned methods model the execution of the application and model the interaction between applications through the exchange of messages.

Network In the network, latency, jitter and packet loss are modelled. Latency can be specified in two ways. The first is called ConstantNetwork, where latency is a constant value for all pairs of processes. The second is called MatrixNetwork, where latency is given by a latency matrix. Jitter can be used with a uniform distribution, a lognormal distribution, or it can be disabled. Packet loss is defined by the percentage of packets lost, from 0.0 (0%) to 1.0 (100%). To determine the latency for a particular message, the Network module provides the `get_latency` method which returns an optional. If the message is lost due to packet loss, `get_latency` returns `None`. Otherwise, the network latency is computed through the latency and the jitter, and it is returned.

Asynchrony In a real execution, a local method call can execute before or after the expected time. This can happen due to the concurrent execution of multiple processes in the same machine, or due to an irregular advancement of the physical clock. Let's call this *process asynchrony.*

Let t_0 be the time at which the method is called for the first time. Due to process asynchrony, the effective time at which the method is executed might be different. The Asynchrony model calculates the time t'_0, which takes process asynchrony into consideration in the following manner $t'_0 = t_0 + async\_factor \times op\_duration$, where $async\_factor$ is an asynchrony factor, and *op_duration* is a configuration parameter. This asynchrony factor corresponds to a percentage and it is randomly generated according to one of these distributions: normal, uniform or Weibull. There is also the possibility of not taking process asynchrony into consideration, in that case the method is executed at simulation time t_0.

Configuration The configuration is described in a YAML file. In the configuration file, it is possible to describe the simulation configuration parameters, and the application ones as well. In the simulation's configuration, it is possible to define the total number of processes (n), the asynchrony and network (latency, jitter, packet loss) parameters, the RNG's seed, among others. It is possible to specify the simulation time at which to save a snapshot of the state of the simulation. In a later simulation, it is possible to load the state. It is possible to provide a new seed for the RNG, in order to load the state multiple times and perform different experiments over that state. There is another configuration aspect, churn. Churn can be defined in the same file as the rest of the configuration, or in a separate file. Regarding churn, it is possible to specify the time at which the event should occur; specify if it is a *join*, *leave*, *fail* or *recover*; specify the number of processes (integer) or the percentage of processes (floating point number) affected by this event. Alternatively, it is possible to specify the identifier of the process, instead of the quantity of processes, by using *leave-id*, *fail-id* or *recover-id*. The difference between *leave* and *fail* is that *leave* is a planned leave, and *fail* is an unplanned leave, i.e. a failure. There is also an *end* event, the execution of which results in the orderly termination of the simulation. These churn events are used to model the life cycle of an application.

Event An event contains:

ts: simulation time at which the event should execute

kind: type of event (*Message*, *Local*, *Churn*, *Save*)

target: specifies in which process the event should be executed

op: operation to be executed at the application level

An event can be of type *Message*, when it is a message to be sent; of type *Local*, when it is a local method call (periodic or not); or a special kernel event, namely an event of type *Churn* or type *Save*. An event of kind *Save* enables saving a snapshot of the state of the simulation in a file. This way, the state can be loaded later on. This makes it possible to run different simulations over the same state.

Simulation Kernel The Simulation Kernel has a structure that contains events, namely a heap. The Simulation Kernel has the responsibility of executing the simulation, handling events in a chronological order. If the next event to execute is a special kernel event, then that event is treated directly by the kernel. Otherwise, the kernel calls the method `receive` of the destination process, which will execute the event's operation in the corresponding application. The kernel guarantees isolation between processes. Unlike what happens in PeerSim, which allows an application to call methods of another application directly, Corten guarantees that the communication between applications is necessarily made through messages, preventing programmer mistakes.

3.2 Random Number Generation

Every simulation tool needs a Random Number Generator (RNG), specifically a pseudo-random generator. A pseudo-random generator produces random values based on an initial value, called seed. Given the same seed, a pseudo-random generator generates the same values in the same order. This is important for simulation because when doing several simulations with the same initial state or parameters, if we use the same seed, we will obtain the same results. This means we can have reproducibility, one of the most important features of simulation. A pseudo-random generator also allows to obtain different results on simulations with the same initial state or parameters, by using different seeds for each simulation. The Rust programming language, which was used for the Corten simulator, contains several libraries for generating random values. From the available RNG, we selected the XorShiftRNG [DD] because it is a pseudo-random generator that is efficient and that can be serialised. When saving a snapshot of the state of the simulation, the internal state of the RNG is also saved. It is advantageous to be able to use, at the application level, the same RNG as the simulator. This way the application not only ensures reproducibility, but it is easy to test the application with different values without changing application code. With this in mind, we provide an API that generates random numbers between 0 and 1 (`get_random()`) and another one (`get_rng()`) that gives access to the RNG itself so that the user can generate other types of values. The user can access these APIs in their applications through the corresponding processes.

3.3 Event Queue Structure

According to [Jai91, pp 408-411], some of the data structures that have been proposed for storing events are: ordered linked list; indexed linear list; tree structures.

In Corten, the chosen structure for the event queue was a min-heap. This was due to its efficiency and scalability. A min-heap is always sorted, and has the element with the smallest key at its root. Access to the smallest element can be done in time $O(1)$. Insertion and deletion operations can be done in time $O(\lg(n))$. The min-heap used in Corten is from a library called `binary-heap-plus` [Sek]. During the course of the work, we contributed to this library, implementing its serialisation capabilities, which it previously lacked. This is important because it is necessary that the event structure, in this case the heap, is serialisable so that the simulation state can be serialisable too.

3.4 Interactivity and Extensions

Besides Corten's functionality of creating a snapshot at a moment in the simulation time specified in the configuration, the user can also create a snapshot at any point in time, through a signal. By pressing ctrl-c on the console, a Corten user triggers the checkpointing feature and the current state of the simulation is serialised to a file. If ctrl-c is pressed a second time before serialisation finishes, serialisation is cancelled and the program is terminated.

The user can create customised versions of network, jitter and asynchrony by implementing the corresponding trait. A trait in Rust is similar to an interface in Java.

3.5 Discussion

With Corten we wanted to have useful functionalities and we wanted Corten to be memory efficient and scale well.

To achieve memory efficiency and scalability, we used Rust as the programming language for Corten. Rust is a system programming language with efficiency concerns and correction guarantees. Rust, by design, has no memory management problems.

Corten supports checkpointing, which is useful to save the state of simulations and reuse them several times in order to perform multiple experiments over the same state. Corten is the only simulator, to the best of our knowledge, that provides process asynchrony, which makes application execution more realistic. Corten supports different types of network latency and jitter, and takes packet loss into consideration.

In the next chapter, we evaluate Corten and showcase its efficiency and performance when compared with a state-of-the-art approach.

Chapter 4

Evaluation

In this chapter, we present the evaluation of Corten. The goals of this evaluation are the following: to demonstrate the correction of different functionalities, through micro-benchmarks described in Section 4.1, and to demonstrate the behaviour in a real application, through the macro-benchmark described in Section 4.2. Each micro-benchmark evaluates a single functionality. The macro-benchmark evaluates the simulator as a whole.

4.1 Micro-benchmarks

In this section, we evaluate Corten's different functionalities through a set of micro-benchmarks. For the micro-benchmarks, we created a simple application which sends *Echo* messages and upon receiving an *Echo* replies an *EchoReply* message. The code for the Echo application is in Appendix A. For this application, there are configurable parameters: *cycles*, *period*, *fanout*. The application has an operation *Cycle*, which repeats *cycles* times, in time intervals of *period*. Each time operation *Cycle* is executed *fanout* echos are sent. In all micro-benchmarks defined next, *fanout* is set to 2, *cycles* is set to 2 and *period* is set to 200, except when other values are explicitly stated. The number of instances of the application, i.e. the number of nodes, is 10.

We start by observing the impact of churn in the processes' life cycle. In Figure 4.1, we can see the churn configuration used for the micro-benchmark concerning churn. The churn configuration lines are of easy interpretation. For example, on line 2, 100% (1.0) of the processes will join the network at simulation time 0. On line 3, the process with identifier 9 will leave the network at simulation time 100. On line 4, 5 processes will leave the network at simulation time 220. On line 7, 50% (0.5) of the processes will be recovered at simulation time 350. In Figure 4.2, we can see the configuration used for the micro-benchmark concerning churn. In this configuration file, the churn configuration file is specified along with seed for the RNG,

```
churn:
  - [0, join, 1.0]
  - [100, leave-id, 9]
  - [220, leave, 5]
  - [250, recover-id, 9]
  - [320, leave, 0.4]
  - [350, recover, 0.5]
  - [500, leave-id, 9]
  - [600, recover, 0.5]
```

Figure 4.1: Churn configuration used in the churn experiment

```
churn_file: config/churn-test.yaml

seed: 0

asynchrony:
  type: NoAsynchrony

network:
  type: ConstantNetwork
  latency: 100
  jitter:
    type: NoJitter
  loss: 0.0

n: 10
fanout: 2
cycles: 2
period: 200
```

Figure 4.2: Configuration used in the churn experiment

asynchrony is not taken into consideration, a network with a constant latency of 100 time units is used, jitter is not taken into consideration, there is no packet loss. The application specific configuration is also specified in this configuration file.

In Figure 4.3, it is possible to see the evolution of the number of processes which are active as the simulation time progresses and, in Figure 4.4, it is possible to see at which points in simulation time the process with identifier 9 is online or offline. It can be stated that the life cycle described in the configuration is observed, both from the point of view of the process and the global point of view of the system.

Next, we study the impact of network latency. In the micro-benchmark concerning network latency, the application is used with just one cycle (*cycles* $= 1$). Two simulations were executed, one in which the network latency was defined as constant and equal to 100 time units, another in which the latency was defined based on a latency matrix. Figure 4.5 shows the percentage of received messages through simulation time, using constant latency and latency matrix. The x axis represents the simulation time and the y axis represents the percentage of received messages until time x. At simulation time 0, each process will send 2 Echo messages. When the Constant

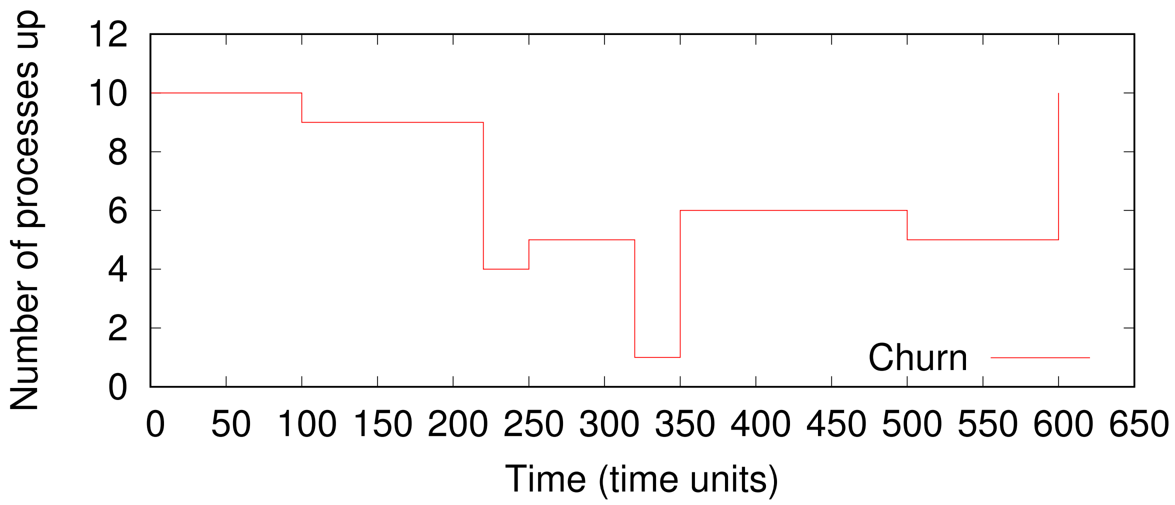

Figure 4.3: Number of processes alive over time

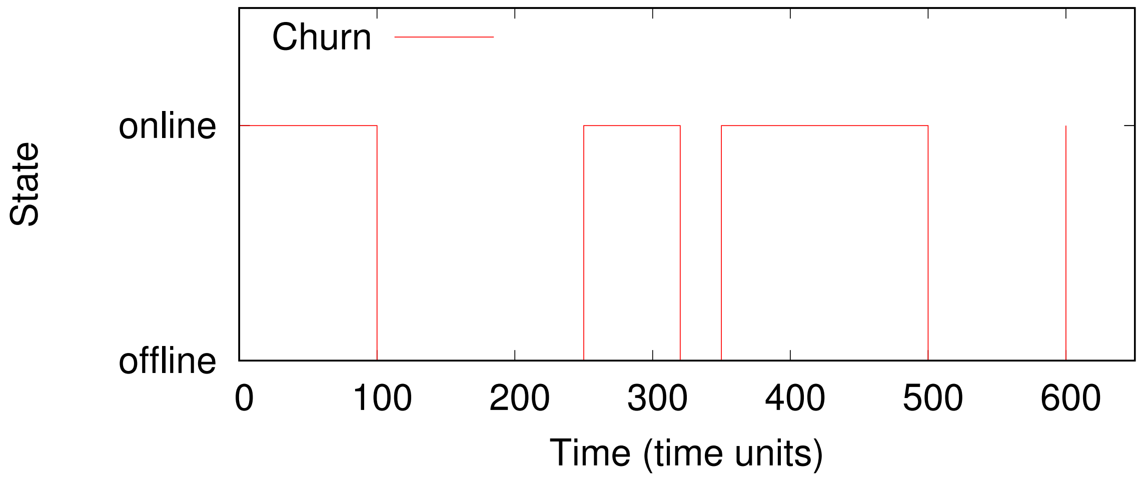

Figure 4.4: Life cycle of process 9

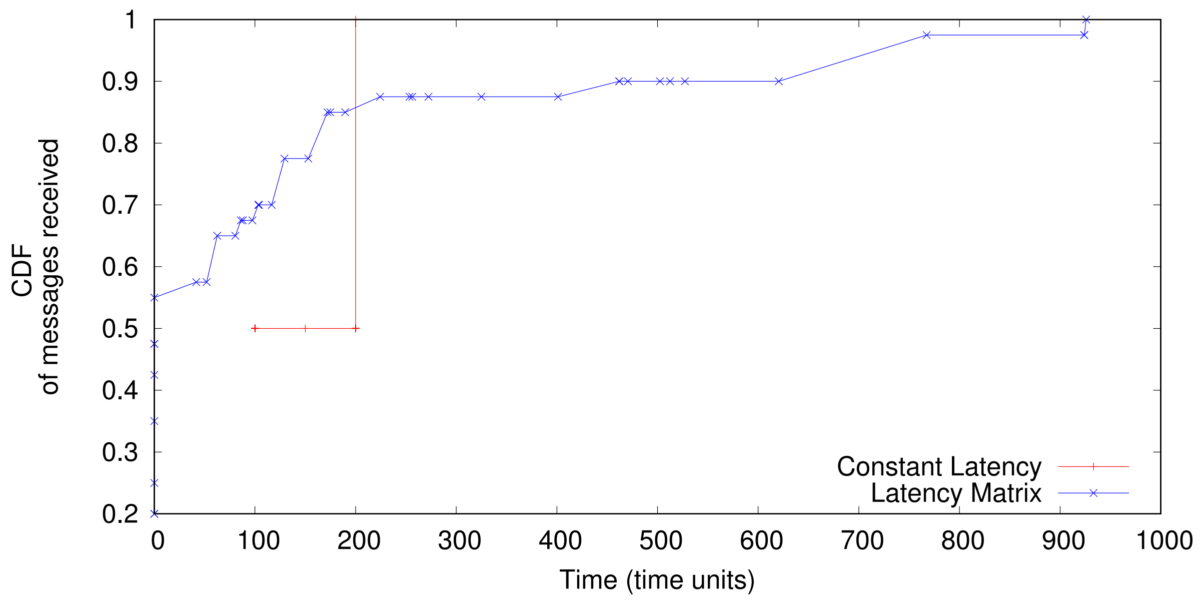

Figure 4.5: Network Latency

Latency Network is used, each message has latency 100. So the Echo messages will arrive at their destinations at simulation time 100. Then, when each application receives the Echo messages, it will send an EchoReply message, which will have latency 100, reaching its destination at simulation time 200. We can conclude that 50% of the messages, i.e. the Echo messages, will arrive at their destination at simulation time 100, and the remaining 50% of messages, i.e. the EchoReply messages, will arrive at their destination at simulation time 200. This is exactly what we can see in Figure 4.5 for Constant Latency. When using Latency Matrix, the latency between any pair of nodes varies as determined by a matrix, and so messages arrive at their destination at different simulation times as seen in Figure 4.5. There are some messages that arrive at their destination at simulation time 0 because they were messages sent from a sender to itself which results in zero latency.

Finally, we evaluate the impact of process asynchrony. In the micro-benchmark concerning asynchrony, the application is used with 10 cycles ($cycles = 10$). In Figure 4.6, the asynchrony values of calls to operation *Cycle* were used, for several asynchrony types (uniform, normal, Weibull, and no asynchrony). The x axis represents the simulation time and the y axis represents the percentage of calls that have an asynchrony value up to x. The uniform distribution is used with a lower bound of -0.1 and an upper bound of 0.1. The normal distribution is used with a 0.0 mean and a 0.1 standard deviation. The Weibull distribution is used with a 1.0 scale and a 1.5 shape. No asynchrony results in an asynchrony value of zero, as can be seen in Figure 4.6. For uniform, normal, and Weibull asynchrony, the asynchrony values vary according to each distribution.

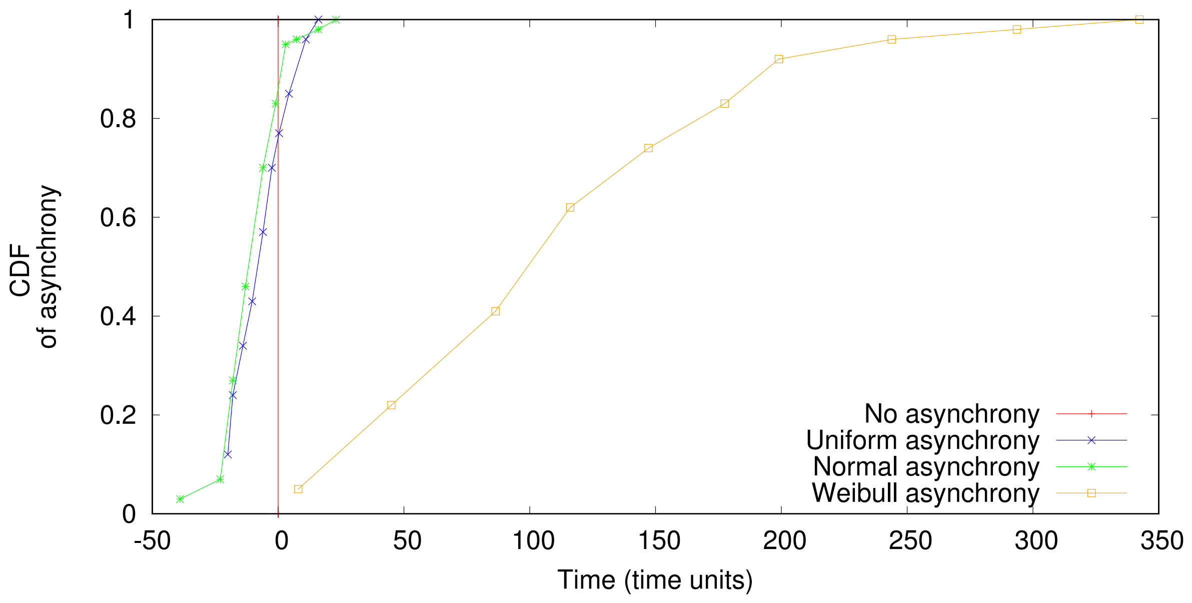

Figure 4.6: Process Asynchrony

4.2 Macro-benchmarks

We now evaluate Corten with a fully fledged application, Chord [SMLN$^+$03], described in the next section. We compare Corten with PeerSim, a popular P2P simulator discussed in Section 4.2.2.

4.2.1 Case study: Chord

In this section, the implementation of Chord in Corten is described along with its main functionalities. Chord is a distributed hash table (DHT). The nodes are virtually arranged in a ring, called the Chord ring, and each node knows its successor, which is the node that immediately follows it; its predecessor, which is the node immediately before; and a set of other nodes called finger table. In our implementation of Chord, we include the optimisation in which a node knows its r successors instead of just one. A ring is said to be stable when all the information of the successors, predecessores and finger table that each node contains is correct. Chord can be used when some content, such as a file, needs to be maintained in a distributed fashion. Each content has an associated key. The content is saved in the node which is the successor of the content's key. When a node wants to obtain a certain content, it looks for the corresponding key in the ring by doing a lookup of the key.

The `leave` method, which every Corten application must implement, allows a node that knows it is going to leave the ring to let its predecessor and successor know that it is leaving, giving them the information of its successor and predecessor, respectively, keeping the ring stable even if a node leaves. Another method that an application implemented in Corten needs to implement is the `on_load` method. Corten has the functionality of saving and reloading the state of a simulation. Combining these Corten functionalities, it is possible to create a Chord ring, stabilise it, save the state and at later point in time reload the state and perform lookups.

4.2.2 Comparison with PeerSim

In this section, we present the obtained results of comparing a Chord's implementation in Corten with a Chord's implementation in PeerSim.

The Chord's implementation in PeerSim used was based on [Tat], with the adjustments described below. In the original implementation, at the start of the simulation an already stable ring with the initial number of nodes was created. This was changed so that there is a stabilisation period and only after that the lookups are performed. This change was made so that Corten's and PeerSim's Chord implementations could be compared.

For each simulator, we evaluate four scenarios, with 1 000, 10 000, 100 000 and 1 000 000 nodes. In each simulation, 10 lookups per node are performed. For Corten, first the Chord ring is stabilised in one simulation and a snapshot of the state of that simulation is saved in a file, then the saved state is loaded in another simulation and the lookups are made. All the discussed Chord simulations' results in Corten take into consideration both the stabilisation simulation and the lookup simulation.

In Figure 4.7 and Figure 4.8 it is possible to see the memory used and the real time a simulation lasted, with 1 000, 10 000 and 100 000 nodes, for both Corten and PeerSim. It is also possible to see the same data with 1 000 000 nodes for Corten. For PeerSim with 1 000 000 nodes and around 16 GB of memory, the simulation stopped before completion due to insufficient memory. The simulation, with 1 000 000 nodes for PeerSim, was repeated with a larger amount of memory available (around 50 GB); it did not finish in 7 days, so it was terminated. Due to the mentioned above, there are no data for 1 000 000 nodes for PeerSim. Note that in Figure 4.7 and Figure 4.8, the x axis and the y axis are in logscale. Each simulation was run 5 times, and the results presented in the time and memory plots are the average of these 5 runs.

As it is possible to observe, Corten's simulations need less memory and take less time to finish than PeerSim's simulations. Corten is more than twice as fast as PeerSim and Corten's speed advantage is even more pronounced with more nodes. For example, with 100 000 nodes, Corten is 2.7 times faster than PeerSim. Corten uses a lot less memory than PeerSim. For 100 000 nodes, Corten uses 14 times less memory than PeerSim. Also Corten with 100 000 nodes can be executed in a personal computer with 8 GB of memory, but PeerSim with the

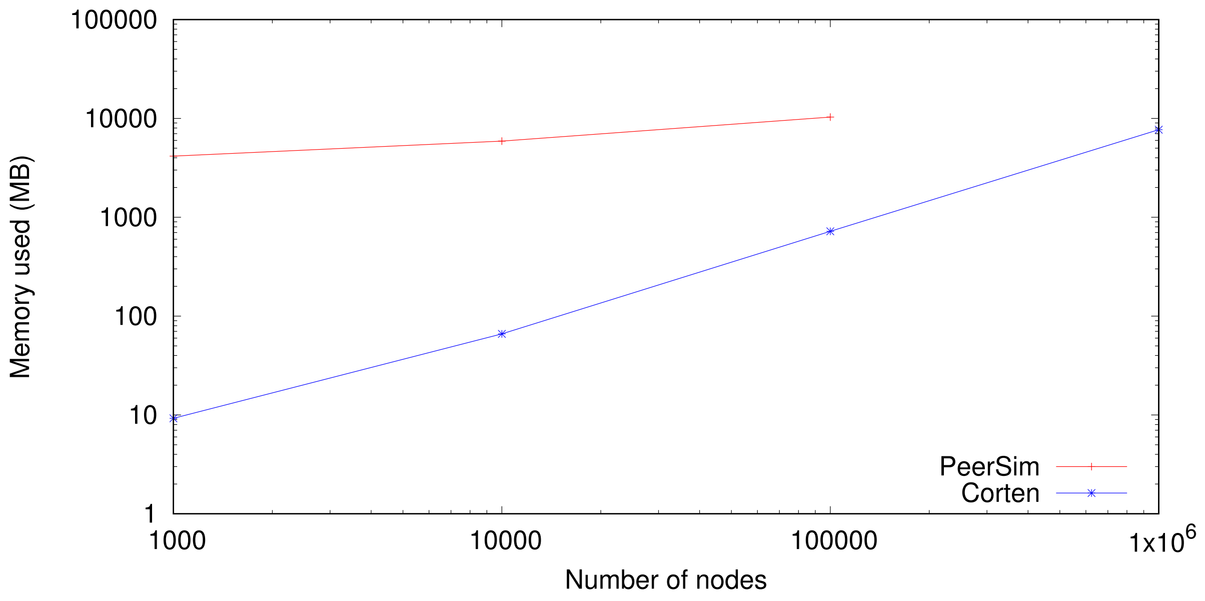

Figure 4.7: Memory (MB) used in Chord simulations, for Corten and for PeerSim. Note the logarithmic scale in the x and y axis.

same number of nodes cannot.

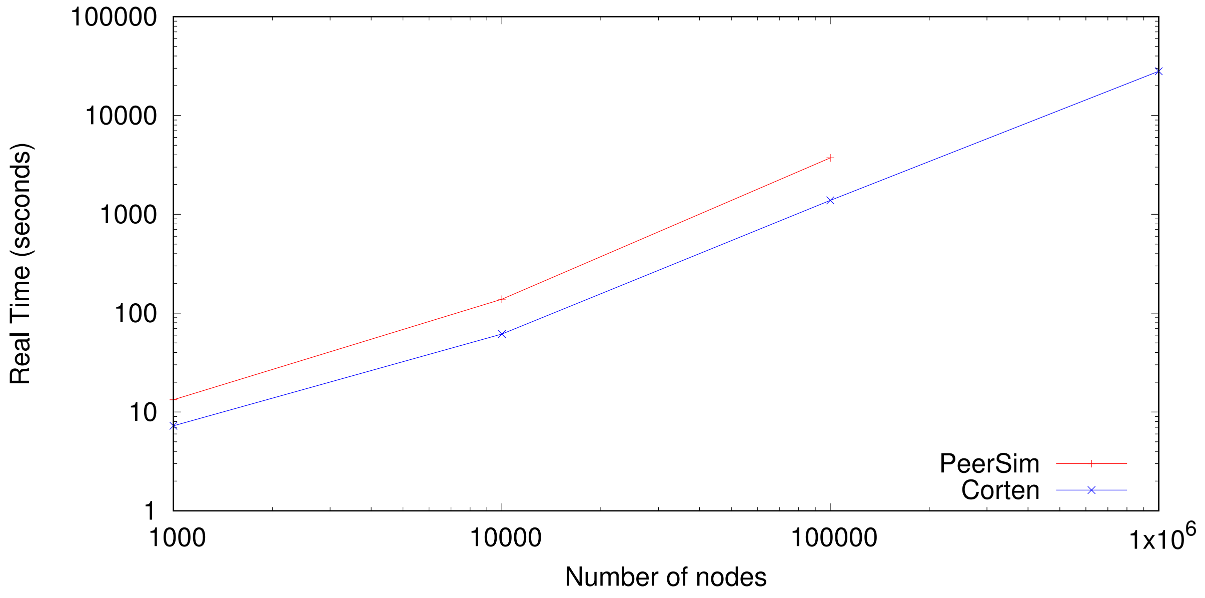

Figure 4.8: Real time (seconds) of the duration of the Chord simulations, for Corten and for PeerSim. Note the logarithmic scale in the x and y axis.

Chapter 5

Conclusions

Distributed algorithms are very important nowadays. It is important to have guarantees that a distributed algorithm works properly. In order to do that algorithms should be tested using tools, such as simulators.

With this work, we set out to create Corten, a distributed algorithms simulator, that would include, in a single tool, all the functionalities that are present in existing simulators and also have a new feature, process asynchrony. We also wanted Corten to be memory efficient and to be able to scale well.

Corten supports checkpointing. A snapshot of the simulation state can be saved, therefore long simulations can be run, it is possible to save a snapshot and to run multiple experiments by loading the same state several times. Corten's simulations are reproducible due to the judicious use of a pseudo-random generator in a controlled way. Corten guarantees isolation between applications. The only way applications can communicate with one another is through messages, preventing unwanted programmer mistakes.

Corten provides an efficient means to test distributed algorithms with thousands of processes. Corten was tested with up to one million nodes. We compared Corten with a state-of-the-art simulator, PeerSim, and Corten scaled better both in terms of time and memory. Corten is more than twice as fast as PeerSim. For 100 000 nodes, Corten uses 14 times less memory than PeerSim.

5.1 Future Work

As future work, Corten can be extended to support parallelisation, in order to take advantage of multiple cores. As a consequence, the real execution time of a simulation can be reduced, which allows for multiple simulations to be executed in the real time interval that previously only one simulation would execute. Also under consideration is the creation of a simplified interface to facilitate even more the development of distributed algorithms in Corten. Presently, a developer that wants to create a Corten application must write Rust code directly. We aim to provide a clean high-level language, so that, in a way similar to that of DistAlgo, a developer can quickly create prototype applications with less code and without having to interact with Rust directly. Furthermore, a future version of Corten may not be only for simulations but also for real executions, by replacing the simulation kernel with a system that allows, transparently, to do the deployment of the algorithms in a real environment.

Bibliography

[CDZ97] K. Calvert, M. Doar, and E. W. Zegura. Modeling internet topology. *IEEE Communication Magazine*, 1997.

[DD] The Rust Project Developers and The Rand Project Developers. https://docs.rs/crate/rand_xorshift/0.1.1.

[DZD+03] F. Dabek, B.Y. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common api for structured peer-to-peer overlays. *2nd International Workshop on Peer-to-Peer Systems, IPTPS'03, Berkeley, CA*, 2003.

[FV01] K. Fall and K. Varadhan. The ns manual (formerly ns notes and documentation). *The VINT Project*, 2001.

[GADK09] Wojciech Galuba, Karl Aberer, Zoran Despotovic, and Wolfgang Kellerer. Protopeer: a p2p toolkit bridging the gap between simulation and live deployment. *International Conference on Simulation Tools and Techniques*, 2009.

[GPM+05] Pedro García, Carles Pairot, Rubén Mondéjar, Jordi Pujol, Helio Tejedor, and Robert Rallo. Planetsim: A new overlay network simulation framework. *Software Engineering & Middleware*, 2005.

[HBH07] D. Hildebrandt, L. Bischofs, and W. Hasselbring. Realpeer – a framework for simulation-based development of peer-to-peer systems. *PDP '07: Proceedings of the 15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing*, 2007.

[Jai91] Raj Jain. *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling.* Wiley professional computing. Wiley, 1991.

[KAB+07] C. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. Vahdat. Mace: language support for building distributed systems. *PLDI '07 Proceedings of the 28th*

ACM SIGPLAN Conference on Programming Language Design and Implementation, 2007.

[KAJV07] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Detecting liveness bugs in systems code. *NSDI*, 2007.

[KHZ^{+}05] Wu Kun, Dai Han, Yang Zhang, Sanglu Lu, Daoxu Chen, and Li Xie. Ndp2psim: A ns2-based platform for peer-to-peer network simulations. *Parallel and Distributed Processing and Applications - ISPA 2005 Workshops*, 2005.

[LCH^{+}05] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. *SIGOPS Oper. Syst. Rev., 39*, 2005.

[LMTY02] Leslie Lamport, John Matthews, Mark Tuttle, and Yuan Yu. Specifying and verifying systems with TLA^{+}. *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, 2002.

[LRF09] Lorenzo Leonini, Etienne Rivier, and Pascal Felber. Splay: Distributed systems evaluation made simple. *Symposium on Networked Systems Design and Implementation*, 2009.

[LSL17] Yanhong A. Liu, Scott D. Stoller, and Bo Lin. From clarity to efficiency for distributed algorithms. *TOPLAS*, 2017.

[MBR03] Gurmeet Singh Manku, Mayank Bawa, and Prabhakar Raghavan. Symphony: Distributed hashing in a small world. *Proceedings of USITS '03: 4th USENIX Symposium on Internet Technologies and Systems*, 2003.

[MJ09] A. Montresor and M. Jelasity. Peersim: A scalable p2p simulator. *International Conference on Peer-to-Peer*, 2009.

[MM00] A. Medina and I. Matta. Brite: A flexible generator of internet topologies. *Technical Report, Boston University, Boston, MA*, 2000.

[MM02] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. *Druschel P., Kaashoek F., Rowstron A. (eds) Peer-to-Peer Systems. IPTPS 2002. Lecture Notes in Computer Science*, 2002.

[RD01] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Guerraoui R. (eds) Middleware 2001. Middleware 2001. Lecture Notes in Computer Science*, 2001.

[Sek] Hideki Sekine. https://crates.io/crates/binary-heap-plus.

[SGR^{+}11] Dominik Stingl, Christian Gross, Julius Rückert, Leonhard Nobach, Aleksandra Kovacevic, and Ralf Steinmetz. Peerfactsim.kom: A simulation framework for peer-to-peer systems. *International Conference on High Performance Computing & Simulation High Performance Computing and Simulation (HPCS)*, 2011.

[SJG17] Shivangi Surati, Devesh C. Jinwala, and Sanjay Garg. Review: A survey of simulators for p2p overlay networks with a case study of the p2p tree overlay using an event-driven simulator. *Engineering Science and Technology, an International Journal*, 2017.

[SMLN^{+}03] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 2003.

[SPS^{+}09] S. Sioutas, G. Papaloukopoulos, E. Sakkopoulos, K. Tsichlas, and Y. Manolopoulos. A novel distributed p2p simulator architecture: D-p2p-sim. *CIKM '09: Proceedings of the 18th ACM Conference on Information and Knowledge 10). Management, ACM, New York, NY, USA*, 2009.

[STS08] K. Shudo, Y. Tanaka, and S. Sekiguchi. Overlay weaver: an overlay construction toolkit. *Computer Communnications*, 2008.

[Tat] Genc Tato. https://github.com/gtato/chordsim.

[UDS01] P. Urban, X. Defago, and A. Schiper. Neko: a single environment to simulate and prototype distributed algorithms. *In Information Networking, 2001. Proceedings. 15th International Conference on*, 2001.

[WI05] H. Wan and N. Ishikawa. Design and implementation of a simulator for peer-to-peer networks: Optimal-sim. *PACRIM'05: Proceedings of IEEE Pacific Rim Conference on Communications, Computers and signal Processing*, 2005.

[ZHS^{+}04] B.Y. Zhao, Ling Huang, J. Stribling, S.C. Rhea, A.D. Joseph, and J.D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications IEEE J. Select. Areas Commun. Selected Areas in Communications, IEEE Journal*, 2004.

Appendix A

Echo Application

```
extern crate serde;
#[macro_use] extern crate serde_derive;

use corten::simulation::{Process, SimulationKernel, ApplicationBase, Operation,
    ProcessId, Time, utils};

use std::any::Any;
use std::rc::Rc;
use std::cell::RefCell;

//holds the application configuration
//each struct has the derive below to automate debuging/printing and serializing
    the sim.
#[derive(Debug, Serialize, Deserialize)]
pub struct AppConf {
    pub n: ProcessId, //number of processes
    fanout: i8, //number of echos to send
    cycles: u16, //number of echo cycles to run
    period: Time //time between cycles
}

//holds the application configuration, read from the yaml file
impl AppConf {
    pub fn new(n: ProcessId, fanout: i8, cycles: u16, period: Time) -> AppConf {
        AppConf { n, fanout, cycles, period }
    }
}

//struct that holds the application state
#[derive(Debug, Serialize, Deserialize)]
```

```
pub struct EchoApplication {
    id: ProcessId,
    cycle: u16, //number of cycles executed
    nb_echos_sent: i32,
    nb_echos_received: i32,
    conf: Rc<AppConf>,
}

//constructor for the Echo Application
impl EchoApplication {
    pub fn new(id: ProcessId, cycle: u16, nb_echos_sent: i32, nb_echos_received:
        i32, conf: Rc<AppConf>) -> Self {
        EchoApplication { id, cycle, nb_echos_sent, nb_echos_received, conf }
    }
}

//ApplicationBase needs to be implemented for every application
//init is called to initialize the application
//recover is called when the application's process recovers from a failure
//derive serialization
#[typetag::serde]
impl ApplicationBase for EchoApplication {
    fn init(&mut self, process: Rc<RefCell<Process>>) {
        //we initialize the application by schedulling a new Cycle
        process.borrow().periodic(Box::new(Cycle {}), self.conf.period, self.conf.
            cycles);
    }
    fn leave(&mut self, _process: Rc<RefCell<Process>>) {}
    fn recover(&mut self, process: Rc<RefCell<Process>>) {
        if self.cycle < self.conf.cycles {
            let remaining_cycles = self.conf.cycles - self.cycle;
            process.borrow().periodic(Box::new(Cycle {}), self.conf.period,
                remaining_cycles);
        }
    }
    fn on_load(&mut self, _process: Rc<RefCell<Process>>, _apps: &Vec<Rc<RefCell<
        Box<dyn ApplicationBase>>>>) {}

    //boilerplate
    fn as_any(&self) -> &dyn Any {
        self
    }
    //boilerplate
    fn as_any_mut(&mut self) -> &mut dyn Any {
        self
    }
}

//Cycle event, the EchoApplication will execute this periodically
//this can also have parameters, see Echo and EchoReply below
#[derive(Debug, Serialize, Deserialize)]
struct Cycle {}

//implement the logic for the Cycle event inside the invoke method
#[typetag::serde]
impl Operation for Cycle {
    fn invoke(&self, app_b: Rc<RefCell<Box<dyn ApplicationBase>>>, process: Rc<
        RefCell<Process>>) {
        //boilerplate, access the EchoApplication state (struct)
        let mut app_borrow = app_b.borrow_mut();
        let app: &mut EchoApplication = app_borrow.as_any_mut().downcast_mut::<
            EchoApplication>().unwrap();

        //if we still have cycle to go, run again
        if app.cycle < app.conf.cycles {
            //select fanout targets to send Echo to
            for _ in 0..app.conf.fanout {
                //get_random return a random process in the system
                let target = (process.borrow().get_random() * app.conf.n as f64)
                    as ProcessId;


                //in every Cycle, send an Echo message to other nodes
                process.borrow().send(Box::new( //this line is boilerplate,
                    //send Echo message with several parameters (encoded in Echo
                        struct)
                    Echo { sender: app.id, target, msg: app.id as i32 * 100, }),
                        target);

                app.nb_echos_sent += 1;
            }
            app.cycle += 1;
        }
    }
```

```
}

//Echo event/message
#[derive(Debug, Serialize, Deserialize)]
struct Echo {
    sender: ProcessId,
    target: ProcessId,
    msg: i32,
}

//implement handling of Echo message in the invoke
#[typetag::serde]
impl Operation for Echo {
    fn invoke(&self, app_b: Rc<RefCell<Box<dyn ApplicationBase>>>, process: Rc<
        RefCell<Process>>) {
        let mut app_borrow = app_b.borrow_mut();
        let app: &mut EchoApplication = app_borrow.as_any_mut().downcast_mut::<
            EchoApplication>().unwrap();

        app.nb_echos_received += 1;
        //reply to the sender with an EchoReply
        process.borrow().send(Box::new( //boilerplate
            EchoReply { sender: app.id, target: self.sender, nb_echoes: app.
                nb_echos_received }), self.sender);
    }
}

//EchoReply event/message
#[derive(Debug, Serialize, Deserialize)]
struct EchoReply {
    sender: ProcessId,
    target: ProcessId,
    nb_echoes:i32,
}

#[typetag::serde]
impl Operation for EchoReply {
    fn invoke(&self, _app_b: Rc<RefCell<Box<dyn ApplicationBase>>>, _process: Rc<
        RefCell<Process>>) {
        //nothing to do
    }
}
```

```
//simulation finished, compute stats
pub fn stats(apps: &Vec<Rc<RefCell<Box<dyn ApplicationBase>>>>) {
    println!("Gathering stats...");
    let mut echos_sent = Vec::with_capacity(apps.len());
    let mut echos_received = Vec::with_capacity(apps.len());
    let mut max_echos_received = 0;
    let mut min_echos_received =  i32::max_value();
    let mut total_echos_received = 0;

    //traverse all the application instances and gather stats
    for app in apps {
        let app_borrow = app.borrow();
        let a = match app_borrow.as_any().downcast_ref::<EchoApplication>() {
            Some(b) => b,
            None => panic!("not an Application"),
        };
        echos_sent.push(a.nb_echos_sent);
        echos_received.push(a.nb_echos_received);
        if max_echos_received < a.nb_echos_received {
            max_echos_received = a.nb_echos_received;
        }

        if min_echos_received > a.nb_echos_received {
            min_echos_received = a.nb_echos_received;
        }

        total_echos_received = total_echos_received + a.nb_echos_received;
    }
    println!("Echos sent: {:?}", echos_sent);
    println!("Echos received: {:?}", echos_received);
    println!("Echos received: min {} max {} total {}", min_echos_received,
        max_echos_received, total_echos_received);
    println!("DONE");
}


//Reads the configuration file and starts the simulation
fn main() {
    //simulation configuration
    let conf_filename = "config/conf-echo.yaml";
    //initialize app configuration
    let app_conf: Rc<AppConf> = Rc::new(utils::yaml_from_file_to_object(&conf_filename));

    //initialize all nodes
    let mut apps = Vec::new();
    for i in 0..app_conf.n {
        apps.push(Rc::new(RefCell::new(Box::new(
            EchoApplication::new(i, 0, 0, 0,
                                 app_conf.clone())) as Box<dyn ApplicationBase>)));
    }
    //run the simulation
    let kernel = SimulationKernel::init(&apps, conf_filename);

    //simulation finished, compute stats
    stats(&kernel.get_applications());
}
```