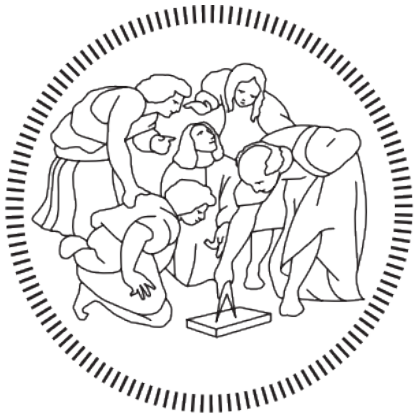


POLITECNICO
MILANO 1863

School of Industrial and Information Engineering
Telecommunications Engineering Course

Data-plane consensus: applying Raft consensus at network level

Supervisor: Prof. Giacomo Verticale

Assistant Supervisor: Eng. Daniele Moro

Emanuele Gallone, Matr. Nr. 914114

Academic Year 2019 - 2020

Abstract

In distributed systems, the consensus problem is one of the most researched topics. Often, applications that require high-availability or fault-tolerance make use of consensus, replicating their state across several remote processes. To maintain consistency however, one must take into consideration that machines will, eventually, fail. The advent of data-plane programming opened new paths to executing custom applications directly at network level, by abstracting network hardware from protocol implementations.

The work presented in this thesis aims at answering the question: can we offload the Raft consensus directly on network level devices, exploiting the programmable data-plane paradigm? We argue for a positive answer while displaying a new protocol and the relative implementation, proposing also a possible use case scenario.

Offloading the Raft consensus algorithm on the network level, can be a possible solution to reduce its cost in terms of overhead and latency, combined with greater throughput. In order to execute Raft on programmable network devices, we use P4-16 as programming language, defining a custom new protocol on top of UDP. We demonstrate the correctness and effectiveness of our protocol implementation, by evaluating an implementation for the BMV2 software switch. We expect that more significant performance gains can be achieved using switches with programmable hardware.

Sommario

Uno degli argomenti di più interesse, in ambito sistemi distribuiti, riguarda il consenso. Spesso, le applicazioni che hanno tra i requisiti quelli di resilienza ai crash, utilizzano algoritmi di consenso per replicare il loro stato su nodi distribuiti nella rete. Per mantenere questi stati consistenti tra di loro, bisogna altresì considerare che i suddetti nodi, eventualmente, possano fallire. L'adozione del data-plane programmabile ha aperto svariati nuovi scenari per eseguire applicazioni direttamente al livello network, astraendo hardware utilizzato in ambito reti dalle varie implementazioni di protocolli. Il lavoro presentato in questa tesi ha come obiettivo quello di rispondere alla seguente domanda: si può implementare l'algoritmo di consenso Raft direttamente sui dispositivi di rete, sfruttando il data-plane programmabile? Tramite questo lavoro siamo in grado di fornire una risposta positiva, mostrando un nuovo protocollo e la sua relativa implementazione, proponendo anche uno scenario d'uso. Applicando Raft direttamente sui dispositivi di rete potrebbe essere una soluzione per diminuire il suo impatto sulle prestazioni in termini di latenza e overhead. Per eseguire Raft su hardware programmabile, abbiamo optato per l'utilizzo di P4-16 come linguaggio di programmazione, definendo un nuovo protocollo sfruttando UDP. Come architettura di riferimento è stato utilizzato BMV2, un software switch sviluppato principalmente per testare nuove applicazioni P4, fornendo allo sviluppatore grande flessibilità a discapito di prestazioni elevate.

The cave you fear to enter holds the treasure you seek.
Joseph Campbell

Contents

List of Figures

List of Tables

Chapter 1

Introduction

When it comes to application availability, users' expectations often fall within the "perfect execution" spectrum, meaning that they expect the services to be available 100% of the time, despite it remains very difficult to provide such up-time. To achieve as much availability as possible, however, services often rely on replication mechanisms.

The CAP theorem states that "between consistency, high availability, and partition tolerance, at most two of them can be fulfilled, in any distributed system" [1]. Usually, high availability and partition tolerance are provided by systems whose primary goal is scalability. Furthermore, to be able to manage different scenarios of partitioning, the systems designers often rely on horizontal partitioning, being one of the most cost-effective choice. Nevertheless, to manage the different replicas in this scenario, the use of consensus protocols is often the best solution.

Resolving consensus issues, however, involves additional overhead for each request, having a great impact on performance; this implies that, typically, consensus is not used within sys-

tems that require high performances. A consensus algorithm must have the following features in order to tolerate the aforementioned failures:

- **Termination**: The overall consensus will, eventually, terminate, agreeing on some value.
- **Integrity**: If every legit process propose the same value k, the consensus terminates by choosing k as final value.
- **Agreement**: Every legit process must agree on a common value.

Distributed consensus has been long associated to the Paxos algorithm, first described by Lamport [2]. It is widely used within production systems, even though its reputation about comprehensibility is really poor. In many occasions, it has been proved to be heavyweight and unreliable in case of scaling-up. Therefore, consensus has been the center of various studies in the last three decades during which, many suggestions arose for optimizations in terms of performances.

We decided to focus on implementing Raft [3] on software-defined network hardware, a newer consensus algorithm presented in 2014, designed to improve the comprehensibility by dividing each phase of the consensus as an independent concept. In order to instruct network devices to handle the Raft consensus, we used P4-16 as programming language. Recent works and advances within the network programmability field, opened new paths for speeding up consensus. Researchers have already exploited this approach in order to achieve better re-

sults and, in general, greater optimization, for data processing systems ([4] [5]).

1.1 Network function disaggregation

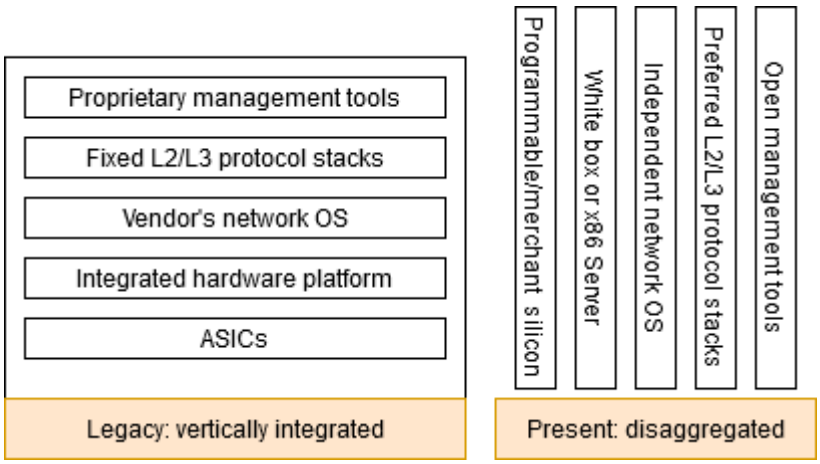


Figure 1.1: Differences between legacy and NFD network equipment.

From an historic point of view, network equipment has always been vertically integrated with products delivered entirely by a single vendor. The purchase of new network hardware has been anticipated by a detailed analysis on its most important characteristics, based on its specific application and architectural location (core, or access layer). Consequently, the network engineers selected the hardware based on the product's provided features and performances among many distinct offerings.

Network Function Disaggregation (NFD) delineates the progression of network equipment from proprietary, closed software and hardware, towards open, modular components which are combined together to build complete network devices, as shown in Figure 1.1. NFD is the evolution from the way the major network devices are designed and built. Disaggregated

devices are integrated horizontally by employing x86 general purpose hardware, ASIC (Application Specific Integrated Circuit) or programmable hardware (such as white-box switches), where the layer 2/3 protocols, network operating systems and network management tools can be independently selected and integrated. The final result will be a unique, well-adjusted device, suited for each specific application [6].

1.1.1 Data-plane programmability

OpenFlow [7] and the advent of Software-Defined Networks (SDN) brought an immense revolution in the way network devices are configured, by defining open interfaces which monitoring or routing application can be built. SDN/OpenFlow helped somehow breaking the network "ossification" by rethinking the networking from a top-down perspective, as stated by Cordeiro *et al.*, in [8]. Nevertheless, OpenFlow does not decouple entirely from the actual protocol implementation, leaving the inability to reshape the switch behavior. Data-plane programmability aims to change dramatically this scenario, by making the switch somehow future-proof against protocols' changes. This approach, however, paves the way to a plethora of possible scenarios, where the network does not only provide connectivity but also services that are transparent to the application level, like in-band telemetry or distributed consensus.

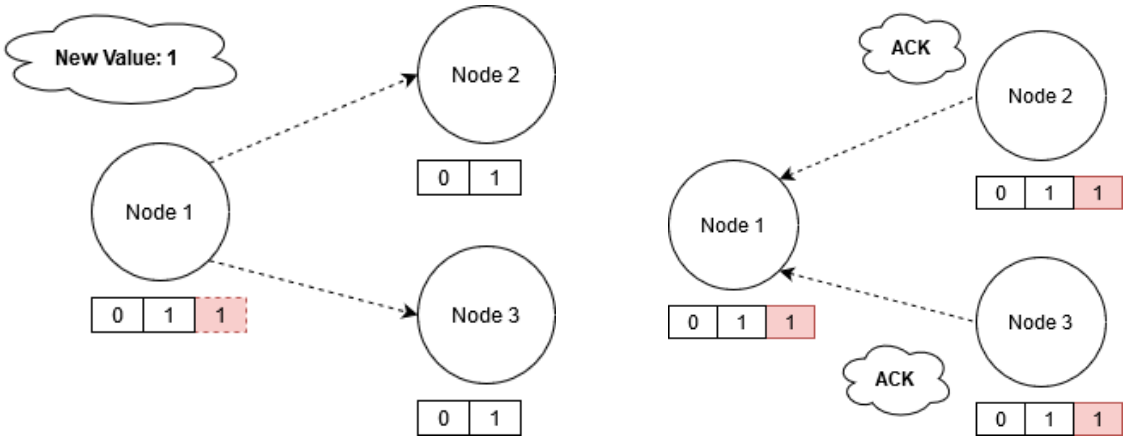


Figure 1.2: Basic example of consensus.

1.2 What does Consensus mean?

Before defining the consensus problem, it is important to distinguish between two types of distributed systems: synchronous or asynchronous systems. In synchronous systems, every process (or node) has access to a global clock, shared among the entire system; this implies that message exchanges are delivered in bounded time intervals. In this scenario, verifying that a node failed is fairly simple: a node is considered faulty if no reply is received from it, in a bounded interval.

On the other hand, having an asynchronous architecture means that no node has any information concerning all the other nodes' speed or state. Messages could be delayed for an arbitrary period of time while clocks may be out of sync, have different precision and so on. In this scenario, there is no way to define bounded time intervals to determine if a node has failed or not.

Consensus means reaching an agreement on a single or multiple values, among remote processes. The higher the number of

nodes that are involved in consensus, the higher the chance that some of them will fail, or worse, start behaving arbitrarily. An example of consensus could be binary consensus, where remote nodes agree on a single value between the set {0,1}. A basic representation is displayed in figure 1.2.

Consensus is a well researched topic [9] and is one of the fundamental problems regarding distributed systems [10], more precisely for high-availability and/or fault-tolerant applications. For example, some of the services at the core of data centers relies on consensus algorithms, like Microsoft Azure [11], Google's Chubby [12] and Zookeeper [13].

Solving a consensus problem requires one or more participants to propose one or multiple values and, at some point in time, a decision will be made, converging to a single value among the entire cluster (*progress guarantee*). Furthermore, once a decision is made, it is final (*safety guarantee*) [2].

An important result has been proved by Fischer, Patterson and Lynch in 1985 [10] about asynchronous distributed consensus; they showed that is not possible to converge towards a single value without making some assumptions about synchronization or reliability. Even a single unannounced process death cannot be tolerated by a consensus protocol, despite the presence of a reliable communication channel.

Solving the consensus problem, however, leads to an inevitable performance degradation [14] due to an higher exchange of messages to maintain consistency among the nodes, creating

many problems for systems that have high-performance requirements.

1.3 Problem description

Implementing and executing a consensus algorithm in a distributed system will most likely lead to an higher latency, as described in the previous section. This is especially true in environments like data centers, where latency is a key metric. The main objective of this work is to reduce the aforementioned latency, exploiting the new paradigm defined by the software-defined networking. Like other several projects, we believe that by bringing consensus logic at a lower level, we can achieve better results while maintaining weak assumptions on the network side [15].

Rate (vps)	Min	Max (ms)	Avg
10	0.4	0.5	0.5
100	0.4	0.5	0.5
500	0.3	0.6	0.4
1000	0.3	0.4	0.3
1500	30.3	637.6	328.1
2000	209.0	3364.2	776.2
2500	436.3	1393.9	898.3
3000	1284.2	8368.4	4967.7

Figure 1.3: *Libpaxos* throughput and latency values.

The latency about the *libpaxos* [17] software library, an actual implementation of the Paxos protocol, is shown within Figure 1.4. Referring to Figure 1.3, to stay in range within data cen-

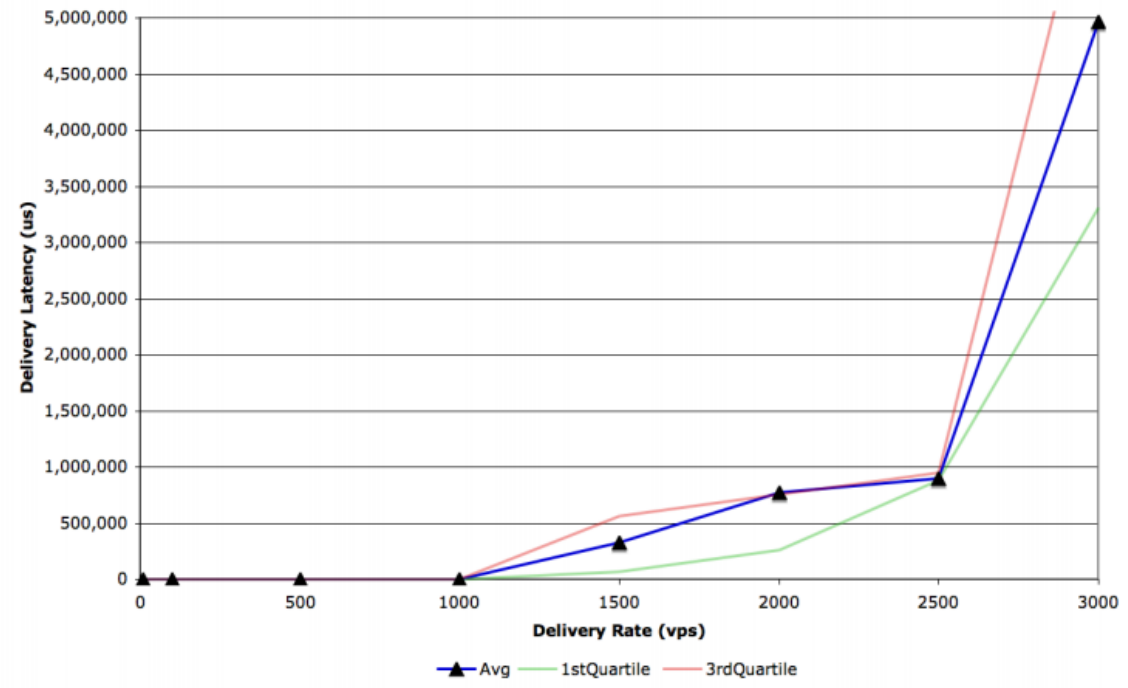


Figure 1.4: *Libpaxos* latency plot. Exceeding 2500 values/s leads to an exponential increase of latency [16].

ter's latency standards, using libpaxos we would be bounded to one thousand values per seconds.

Despite the performance motivations, perform network offloading regarding consensus, allows us to exploit the properties of the network [18], since the consensus itself is strictly correlated to network assumptions, like Lamport described in [19]. Bringing the consensus to the network level, thus, improves the amount of services that the network itself is capable to offer, rather than simple connectivity. Consensus at network level does not require additional hardware (e.g. hardware accelerators) but using, instead, the same devices that are needed to perform forwarding operations to serve connectivity.

1.4 Use case and results' anticipation

As an example, we propose a the load-balancing based on consistent hashing use case. The proposal will be described in details by providing an example on how to implement such scenario using the Raft P4 implementation presented in this work.

1.4.1 Brief results' anticipation

Implementing Raft on network level can be a possible solution to achieve an higher throughput. Being the implementation built on BMV2 [20], an high-flexibility software switch used mainly to debug and develop P4 applications, performance in terms of throughput remains limited between 3-4 thousands of requests/s. We expect that significant performance gains can be reached by using production-grade switches.

1.5 Thesis Overview

This thesis is organized into five chapters. Some information needed to explain some concepts in a later chapter is occasionally referenced to an earlier chapter, to avoid excessive redundancy.

- Chapter two describes the state of the art, regarding topics concerning our work. We review some scientific articles by justifying how they influenced our work.
- Chapter three provides a brief description regarding the consensus problem, how it concerns distributed systems

in general and a possible use case within a data-center environment.

- In chapter four we describe our solution in details, specifying the protocol we developed, analyzing also the limitations.
- Finally, in chapter five and six we discuss the results obtained through our solution and we draw our conclusions, respectively.

Chapter 2

State of the art

2.1 Data-plane programming

Traditionally, the network has been considered as something fixed, where only 'simple' operations can be made, for example forward an IP packet or perform some filtering. In this section, we are going to illustrate some of the new concepts that were and are being developed to enhance the networking. Today, after the standardization of open protocols like OpenFlow[7] and the development of programmable network equipment we can now benefit from the abstraction between hardware and software, the very same that characterize the CPU and GPU worlds, bringing winds of change about the applications that can be realized directly on network level.

OpenFlow has enabled the programmability about the control-plane, by giving more flexibility to network operators. On the other hand, this approach has left the data-plane somehow 'fixed' to protocols defined within OpenFlow. Truly data-plane programmability has no such limitations, meaning that the operator can program the control-plane and the data-plane independently, even by defining its own protocols and syntax

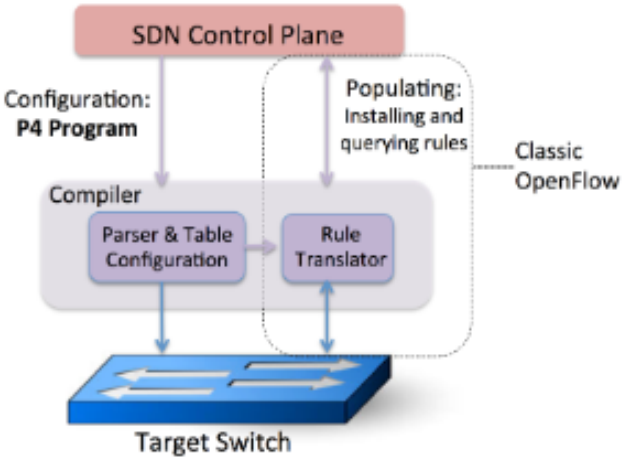


Figure 2.1: Differences between P4 and OpenFlow approaches to data-plane programming [21].

[21]. Therefore, the programmability introduces the capability of a router-switch to display its processing logic to the control-plane, to be, eventually, rapidly re-configured [22]. This results inspired works like the one did by Bosshart, Pat and Gibb about re-configurable match table (RTM) architecture, proving that data-plane programming is possible even in ASICs [23].

Another advantage of data-plane programming is the ability to provide rapid protocol prototyping. Large corporations that manage various data centers, like Google or Amazon, can design their own network hardware and protocols to perform many optimizations concerning their use cases.

There are, however, several problems that are currently being researched, regarding data-plane programming, like the ones discussed in [22]:

- **Performance or flexibility, choose one**. Usually, regarding data-plane programmability, there is a trade-off between performance and flexibility [22]. As an example,

pick the BMV2 [20] and OpenVSwitch [24] implementations: The development of BMV2 switch is focused on flexibility, while OpenVSwitch is a production-grade switch, meaning that it has been optimized for high performance. BMV2 has many useful functions that we used for our implementation (e.g. *clone* function, to clone a packet, useful for propagating information to the control-plane), while OpenVSwitch has not.

- **Non-programmable components**. There are still some components that are not programmable, at all. For example the physical layer is still bounded by hardware. While it provides a full set of services useful for the upper layers, like bit synchronization, line encoding etc., it still is an opaque component, meaning that a data-plane developer will not be able to access it [25].

2.1.1 P4-16

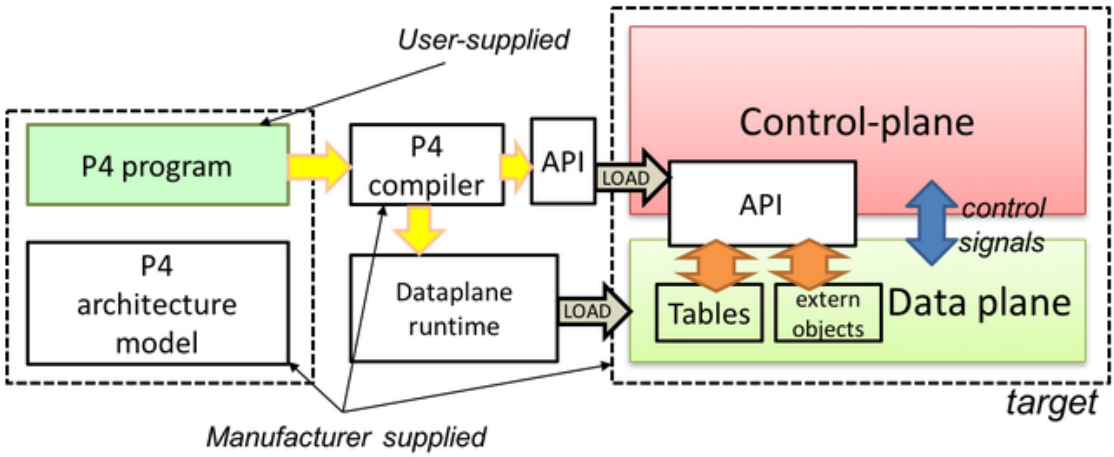


Figure 2.2: P4 Scheme overview [26].

P4-16 (more in general, P4) [27] is a new domain-specific lan-

guage, optimized for network data forwarding, protocol and target independent which provides an abstract model suitable for programming the network data-plane. Figure 2.2 is an high-level representation about the workflow in P4-enabled hardware. Devices that support P4 are protocol independent, meaning that they have no inherent support for any protocol at all. P4 programs enable the device to 'understand' that particular protocol, by defining packet headers, specifying packet parsing and the relative processing behaviors. in P4 programs the developer can define:

- **Headers**: by specifying field name and relative bit width of one or more packet headers.
- **Metadata**: it provides a packet-specific state, with per-packet scope. When a packet ends the processing, the relative metadata is then lost.
- **Registers**: useful when some information must be persistently available.
- **Counters & Meters**: As registers, they are persistent. They can be read by the control-plane, to perform some statistics.
- **Tables & Actions**: tables are used to specify on what packet header field or metadata perform matching (either exact, ternary or longest-prefix). In case of hit (or miss), some actions, defined by the developer, will be executed. The tables can be populated at run-time using *P4Runtime* [28].

- **Apply Section**: also known as *Control section*; it is possible to specify what tables to apply, using, eventually, conditional statements. It is important to remark that the control section is the only area where conditional statement can be made, in the overall P4 program.

Before executing a P4 program on a target switch, a compiler is needed to map the target-independent protocol description on top of the target specific hardware, hence, allocating the needed resources and generating a configuration for the target device. The P4 compiler [29] supports now several back-end targets, like BMV2 [20], developed by Barefoot Networks for the software-based *simple switch* [30].

Generally speaking, P4 applications are not portable across different architectures. For example, a P4 program that forwards packets by writing into a custom register will not work properly on a target that does not provide any function to read/write the register. To cope with this limitation, the PSA (Portable Switch Architecture)[31] is being developed by Barefoot Networks. P4 applications developed using PSA as reference, will be fully portable among devices that implement the aforementioned architecture.

2.2 Consensus Algorithms

Before proceeding with the illustration of our work, we are going to present a quick review about the main algorithms that were developed in the past to solve the consensus problem, among distributed systems. A preliminary step, before

describing a consensus problem, is to make some assumptions about the behaviors of a faulty process, since it affects the complexity of the algorithm by imposing more restrictions. Table 2.1 describes various models, that have been proposed by M. Barborak et al. in 1993 [32].

Table 2.1: Failure models summary.

Type of failure	**Brief description**
Fail-stop	Process terminates; the cluster is aware of the failure.
Omission	A process, for example does not answer back. to some messages
Computation	Process output mismatch (e.g. calculation of a floating point value, due to different CPU
Timing	Process does not perform a task within a bounded time interval.
Authenticated Byzantine	A process can behave maliciously or arbitrarily, being controlled by the adversary.
Byzantine	All the other kinds of failure.

Concerning our work, one of our main assumptions is that we expect the P4 application to be executed within a data center environment, meaning that the security and authentication parts will be delegated to other actors that won't be treated within this work. Therefore, we will consider only the *Fail-stop* failure model, referring to table 2.1. For the sake of completeness, however, we'll also review a consensus algorithm that works under the Byzantine failure model.

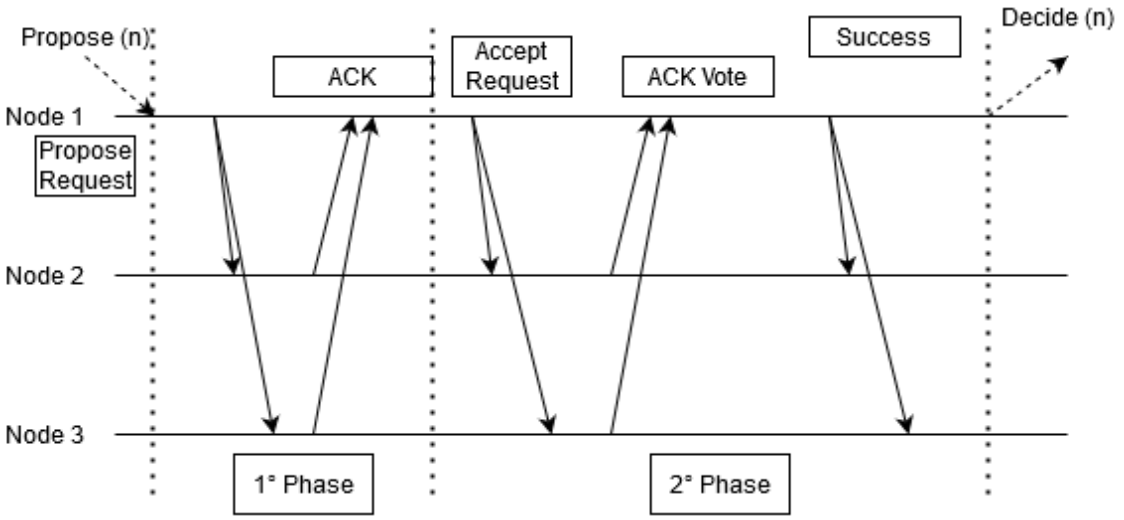


Figure 2.3: Paxos phases

2.2.1 Paxos

Paxos represents, nowadays, a family of algorithms, since many variations have been developed during the years. The original Paxos protocol [2] was first described by Leslie Lamport using the legislators' voting process analogy, on the Paxos island, being the latter not always present inside the Chamber, in ancient Greece. Some years later, Lamport reviewed the protocol, in the "Paxos Made Simple" [33] article, introducing it as Basic Paxos.

In Basic Paxos, every node can be either a *proposer, acceptor* or *learner* and communicate with other nodes through messages.

- **Proposer**: In a single Paxos round, multiple proposers can exist but the protocol will guarantee that a single value will be chosen, at the round's end. It tries to become leader by choosing a random number that will be used in the first phase of the Paxos' round.

- **Acceptor**: once voted for a single value, it will remember it, by storing it. Moreover, the acceptor must not accept other values if it has already voted and accepted one.
- **Learner**: once a value has been accepted by the majority of acceptors, they will, eventually, learn it by receiving the accepted value through messages, by the acceptors. In case of failure, it will retrieve the chosen value by querying the acceptors' quorum.

Figure 2.3 illustrates a successful, full round of the Paxos algorithm. The first phase is further divided between Phase 1A and Phase 1B. In Phase 1A all the proposers will choose a random number, denoted as *round number* and send a *Proposer Request* to a majority or all of the acceptors. In Phase 1B, instead, the acceptors will answer with a *Promise message*, promising that it will reject any other request from other proposers with a round number smaller than the one they received. When a proposer receives *Promise messages* from the majority of the acceptors, it will then conclude the first phase by starting the second one.

In Phase 2, the proper value consensus takes place: the proposer will propose a new value, if the majority of the acceptors has accepted any value, by sending an *Accept request* with the round number it used during Phase 1. Unless another value has already been acknowledged, the acceptor, upon receiving an *Accept request*, will answer with an Acknowledge message, denoted as *Accepted message* to learners. Once the quorum of the acceptors send an Acknowledge message, the round ends,

meaning that consensus has been reached.

Paxos tolerates up to k failures, being $n = 2k + 1$ the total number of acceptors. To guarantee the Liveness property (i.e. the guarantee that the system will make progress), the quorum of the acceptors ($k + 1$) must be non-faulty [34].

2.2.2 Raft

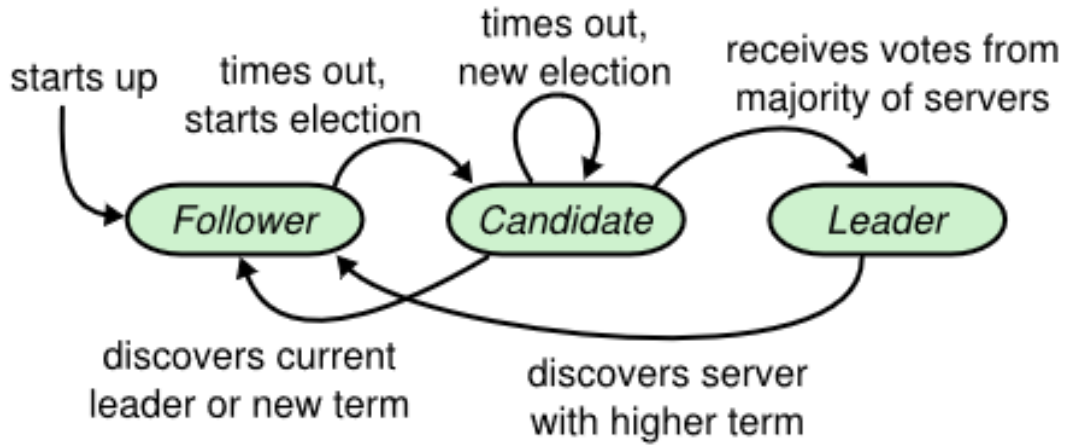


Figure 2.4: Raft states scheme [3]

Raft [3] is a newer consensus algorithm, designed specifically to be more understandable, with respect to Paxos, by separating key concepts of consensus such as leader election and log replication.

- **Leader Election**: When a leader crashes, a new election will occur, ending with the definition of a new leader node.
- **Log Replication**: Once elected, a leader is then responsible to accept new log entries and replicate them across the rest of the cluster.

- **Safety**: when a server has appended an entry to a given index, no other server will apply a different entry for the same index, guaranteeing consistency among distributed nodes.

At any time, a node can either be a *Leader, Candidate or Follower*, as described within figure 2.4. A normal execution of Raft expects only one leader node while all the others are passive followers. Time is divided between *terms* of arbitrary lengths, numbered as an increasing sequence of integers. Referring to figure 2.4, when starting a new Raft cluster, all nodes will begin as *followers*; since every node has an internal timer, at some point in time, one of the nodes will timeout, starting the leader election, thus, changing its own state to *candidate*, seeking votes from the rest of the cluster.

In order to maintain its status, a leader uses the heartbeat mechanism, i.e. sending periodic messages to follower nodes, triggering, hence, their timer to reset, excluding a new leader election. Concerning the consensus part, the leader node sends an *Append Entries* message, waiting for the other nodes to write their own log index and answer back. If the majority of the nodes answer positively, the leader consolidates the new entry notifying the result [3]. Like Paxos, Raft tolerates up to k failures, being $n = 2k + 1$ the total number of nodes within the cluster.

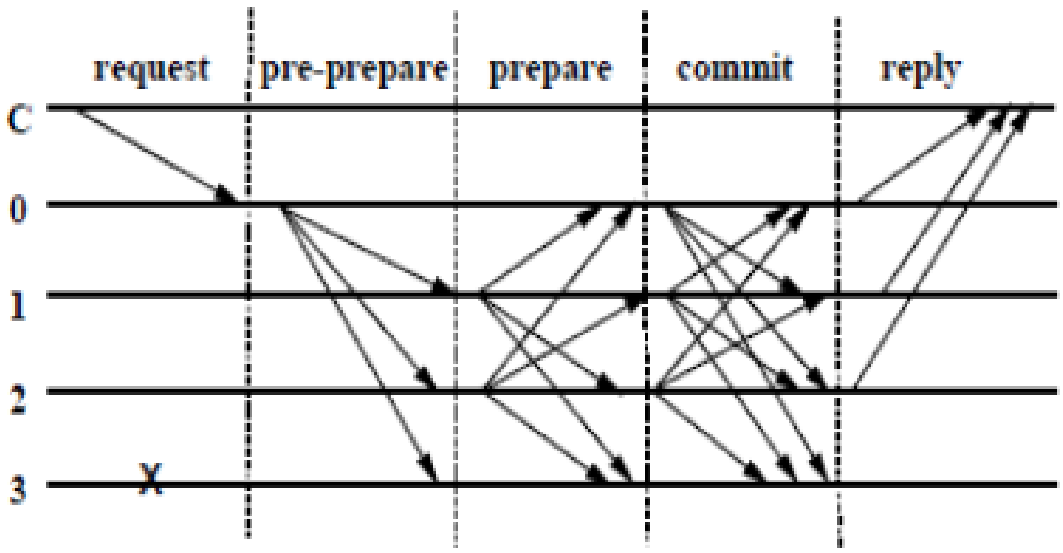


Figure 2.5: PBFT successful consensus scenario [35]

2.2.3 Practical Byzantine-fault-tolerant protocol

The first protocol that considers byzantine failures as fault model, in weakly synchronous environments, is the practical byzantine fault tolerant protocol (PBFT), designed by Liskov and Castro, published in 1999 [35]. Out of K nodes, PBFT can support at most $\left\lfloor \frac{K-1}{3} \right\rfloor$ failures [35], while it makes the assumption that at most $\frac{1}{3}$ of the nodes is malicious; in case this assumption does not hold, the cluster becomes vulnerable to attacks.

Referring to figure 2.5, the PBFT protocol consists of five steps:

- **Request**: The sequence starts with the client sending a request to the cluster's master.
- **Pre-prepare**: while saving the request message and assigning an order number, the master broadcasts to the rest of the cluster a *Pre-prepare* message. The following nodes will determine whether to refuse the request or not.

- **Prepare**: By receiving the *Pre-prepare* message from the master, every node checks its validity adding the message to its local log and multicasting a prepare message to the rest of the cluster showing the it has received a new proposal and accepts it.
- **Commit**: Upon collecting a number of prepare messages of $2K + 1$ that match the pre-premessage order number, every node broadcasts the commit message entering the commit phase. Furthermore, a node needs to receive a quorum of commit messages to ensure that the proposal made by the master node has been accepted and replicated among the majority of the cluster.
- **Reply**: The final phase where the client that started this process receives a number of replies from the majority of the nodes. In case the client does not receives the replies, for example in case of packet loss, the cluster only needs to re-send those replies.

2.3 Related Work

This section provides a list of papers that are related to this thesis topic. These reviews were the preliminary studies to find ideas regarding this thesis. A short description with its limitations is presented, for each paper.

2.3.1 P4xos

In recent times, several projects have been developed, trying to investigate the possibility to leverage the network programmability to enhance application performances.

P4xos [18] is one of the first attempt to bring consensus logic at network level. Like us, the authors of P4xos thinks that we can achieve better results in terms of latency and throughput, by performing network offloading about consensus algorithms. Moreover, the article describes an implementation of the Paxos algorithm using P4, exploring a wide area of problems and design decisions that have not been addressed before. For example, being a router-switch not capable of creating new messages, the Paxos logic had to be mapped 1:1 with routing decisions.

One limitation of this work, with respect to ours, is that the roles onto the switches are hard-coded, meaning that little to no flexibility is provided, about the consensus. Furthermore, if the *coordinator* node fails, the entire structure fails, leading to an impossibility to executing the consensus protocol.

2.3.2 SC-BFT

While P4xos and our work works for crash faults, or generally speaking, non-byzantine faults, SC-BFT (Switch-Centric Byzantine Fault Tolerant) [36] does. It is a new approach to handle byzantine faults within SDN environments. Similarly to our work, they implemented the application through P4, ex-

ecuting it in BMV2 [20] software switches. SC-BFT influenced our way to design the switch controller, i.e. to be 'universal' and not bounded to some Raft's roles. It simply will handle some messages but more importantly, the timings, as it will be explained in details, in the next chapters.

Chapter 3

Proposed solution and use case

3.1 Proposed solution

In this thesis we propose an implementation of Raft Consensus Algorithm [3] developed with P4-16 [27] that can be executed by programmable network devices, paired by the corresponding controller. With respect to the original Raft paper [3], we will introduce some minor modifications to the protocol, mainly because of the non-Turing complete nature of P4-16. We chose P4-16 among the various languages because we consider it more mature with respect to the alternatives. All the code relative to our work is publicly available [1].

Before diving into the implementation we want to discuss about some choices that we took to cope with some limitations. As P4-16 does not handle timings we decided to keep time-based events on the control plane (i.e. the Raft controller, defined within the *Controller.py* script), shown in Figure 4.2b. The interaction between the P4 node and the relative Raft controller is made by *Packet-in* events. The controller implements

[1] https://github.com/EmanueleGallone/RaftP4

a packet sniffer using the scapy library [37].

In order to have an application as router/switch-agnostic as possible, instead of using IP, we decided to use a custom forwarding mechanism based on Raft IDs that are assigned to each device by a bootstrap and configuration server that is responsible of the initialization, by making use of the network automation tool that we developed [2]. This approach allowed us to bypass problems linked to router interface failures, since a node must be univocally identified within the Raft cluster.

By contrast to the implementation of P4xos [18], we built our P4 application taking in consideration the dynamics that characterize Raft; instead of having each node with hard-coded roles, we made the application handling the leader election, thus every node can be elected anytime, during the execution.

3.2 Solution implementation

3.2.1 Packet header

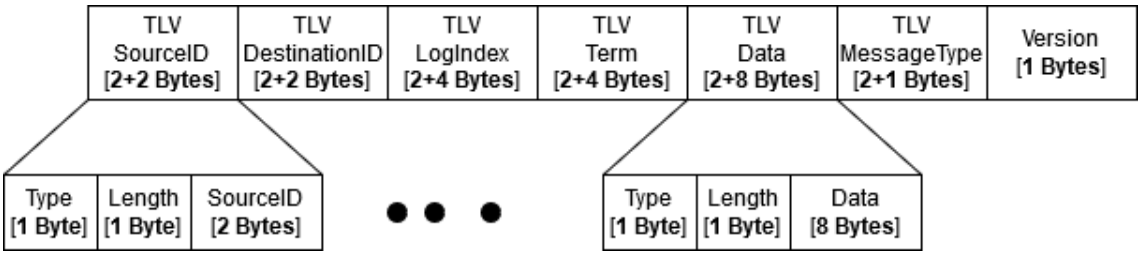


Figure 3.1: Raft on P4 header definition. Every field is defined as a TLV field, except the *Version* field.

We defined a custom header on top of UDP, since we believe that consensus does not justify the complexity that would be

[2] i.e. Switch_Register_Manager.py

introduced to manage TCP traffic. Moreover, being Raft designed to recover from destructive events like crashes and link failures, we can safely assume that the application can be executed even in presence of packet loss.

As described above, to cope with the limitations of P4-16 we had to define a message type header field, since various message types had to be defined. To make the protocol extensible, each field of the header is defined as a TLV (Type-Length-Value) field, as described in figure 3.1. Nevertheless, we had to declare the *value* field with fixed length, because of P4's current limitations:

- ***varbit* field limitation**: P4 provides a *varbit* type to handle variable length header fields (e.g. IPv4 options); unfortunately this comes with a serious limitation: within the P4 application it is not possible to change the actual field value, as specified in Section 8.8 of [26].
- **P4 register bit-width**: To declare and use a register in P4, the aforementioned must be of a well-known size. The P4 compiler does not allow to declare registers of dynamic sizes.

The use of TLV fields allows, however, the protocol to be more flexible. The TLV could be exploited, for example, by transmitting 4 values of 16-bit in a single Raft transaction, using the 64-bit wide *Data field* and encoding the structure within the relative *Type field*, .

Table 3.1: List of message types and respective description.

Message type	Value	Brief description
Heartbeat request	0x1	Used within the heartbeat sequence.
Append entries	0x2	Check if new request can be replicated within the cluster.
Heartbeat response	0x3	Used within the heartbeat sequence as response.
Request vote	0x4	Candidate node requests vote to become leader.
Positive response vote	0x5	Node acknowledge the requester to become leader.
Commit value	0x6	New value can be consolidated within the cluster.
Append entries reply	0x7	Response to append entries. If enough replies are collected, a commit value will succeed.
Recover entries	0x8	Used in recovery mechanism. Recover a single value.
Commit value ACK	0x9	New value has been correctly replicated within a node's log.
Timeout	0xA	Raft controller informs its relative P4 node that a timeout has occurred, starting the leader election.
Negative response vote	0xB	Node replies negatively to vote.
Reject new request	0xC	Node is in transaction mode. New request has been rejected.
Retrieve log	0xD	External client trying to read the Raft log.
Redirect	0xE	New request has reached a non-leader node. Used to redirect the message to leader.
Start heartbeat	0xFE	Raft controller informs the relative P4 node to start the heartbeat sequence.
New request	0xFF	Append new value inside Raft's log.

3.2.2 Message types

To cope with the limitations introduced by the *match-action* abstraction, we had to introduce a variety of message types. Table 3.1 lists all the message types we defined within our solution.

The main reason behind the definition of many message types is easily explained: our protocol is based on the idea of packet ordering, meaning that some operations are a sequence of ordered packets (e.g. a switch would never receive a *commit log* message before the *append entries* message). Since every part of the processes is designed to be as independent from each other as possible, the recovery from disruptive events like packet loss, application is designed to recover from such events; for example, if a node does not receive an *append entries* message, it will in any case commit the new value upon receiving a *commit log* message.

3.2.3 API

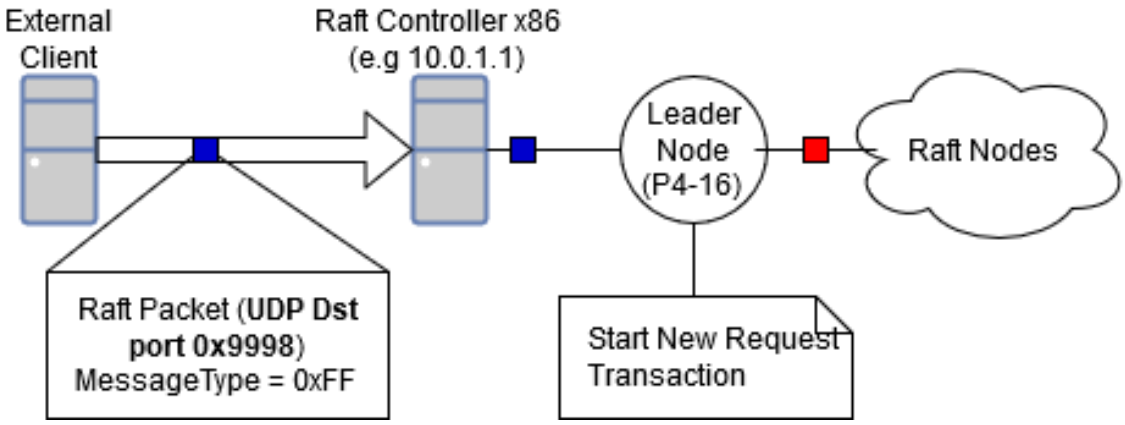


Figure 3.2: External client interacting with the Raft cluster.

Providing an API is fundamental, thus allowing external clients

to interact with the Raft cluster. To reach the service, a client has to send a *new request* message, which carries the value to be replicated within the cluster. We implemented a transaction mechanism to prevent new request overwriting. Without the transaction mechanism there would be an overwrite of the procedure, in case of new request arrivals before the precedent was completed. Therefore, the system could reject the new request by answering negatively with a *reject request* message.

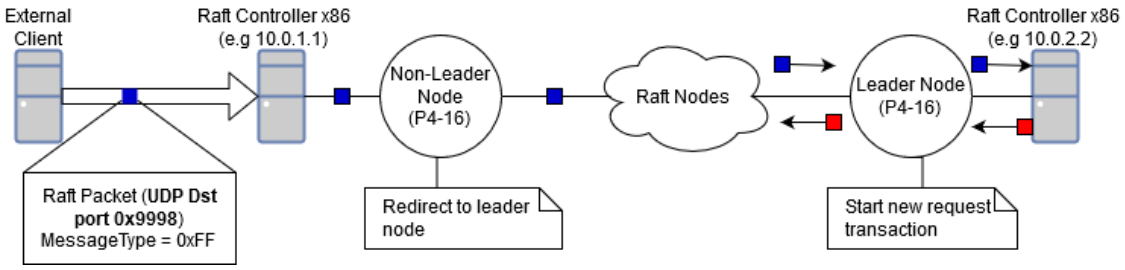


Figure 3.3: New request redirection.

To start the replicating process, an external client sends a *new request* (identified with *messageType* equal to 255, decimal, as showed in table 3.1) Raft packet, with UDP destination port 39320, to any Raft controller. Depending on the node state, a new transaction or a redirect will take place:

- **New transaction**: the external client manages to send the new request to a Leader node. At this point, the leader will apply all the actions that refer to a new transaction. Once terminated, the controller will inform the external client about the successful handling of the new value to append within the log.
- **Redirect**: the external client sends a *new request* message to a Raft controller whose state is not leader. The

relative P4 node will apply the redirect table to forward the request to the leader node(if elected). The redirect workflow is represented within figure 3.3.

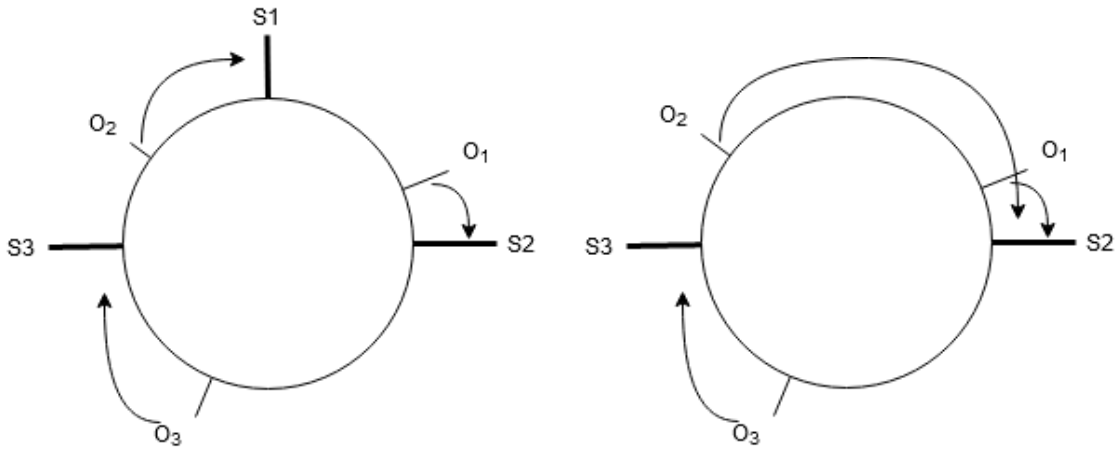
3.3 Use case example

This section will provide a use case's detailed proposal, involving the solution we provide within this work. We found interesting evaluating the case of load balancing performed with consistent hashing.

3.3.1 Load balancing based on consistent hashing

Consistent hashing is a distributed hashing scheme [38], that operates independently of the number of servers. A basic representation of a single hash table is showed within figure 3.4. One property of the consistent hashing is the load balancing; in fact, the hash table contains either the servers' hash and the objects' hash. The outcome is a flexible data structure that allows the balanced mapping between objects and servers, even in presence of horizontal scale-up. The use of consistent hashing implies that the overhead for creating and establish a new TCP connection is paid only once, with respect to simple hash-based load balancing.

A possible implementation of this scheme, using our P4 application, can be made by creating specific actions that will read and write the *logValueRegister* (they will serve as *wrapper* functions, using an analogy to object-oriented programming), instead of directly accessing it using the extern functions pro-



(a) Load balancing by mapping every object to a server, in clockwise order.

(b) Server 1 failure implies its removal from the data structure. Only O_2 is reassigned, keeping the rest of hash table untouched.

Figure 3.4: Hash table representation with 3 servers.

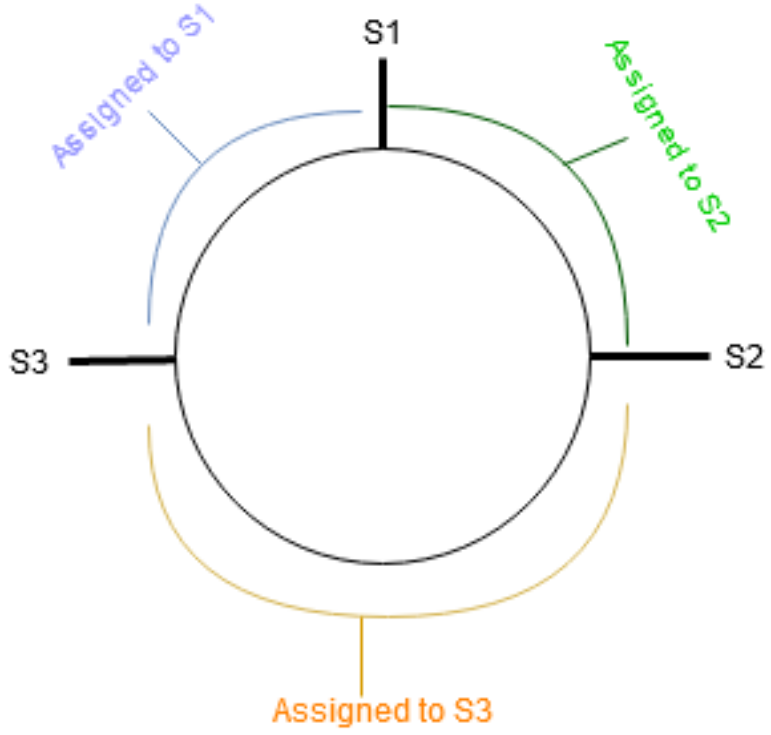


Figure 3.5: Range based mapping. All the requests' hash that resides within a colored area are assigned to the respective server.

vided by BMV2's model. An example of such definitions is provided within Listing 3.1. Using the *modulo* operator allows to treat the log as circular memory. Furthermore, a new entry within the Raft metadata must be defined, that is *logValue*, since P4 actions are not ordinary functions that provide a return value.

```
1  action ReadLogRegister(bit<32> index){
2      logValueRegister.read(raft_metadata.logValue,
3      index % raft_metadata.hashTableSize
4      ); //function read(return value, index)
5  }
6
7  action WriteLogRegister(bit<32> index, bit<64> value){
8      logValueRegister.write(index % raft_metadata.hashTableSize,
9      raft_metadata.logValue
10     ); //function write(index, value to write)
11 }
```

Listing 3.1: 'wrapping' actions to implement the hash table onto the Raft log. Using this approach, the register is abstracted from direct read or write, allowing the use of circular memory.

In case of server failure, the utilization of the watchdog mechanism is recommended. The latter will be in charge of informing the Raft cluster about eventual failures, updating, thus, the hash table saved within the Raft log; an example of this scenario is represented in figure 3.6.

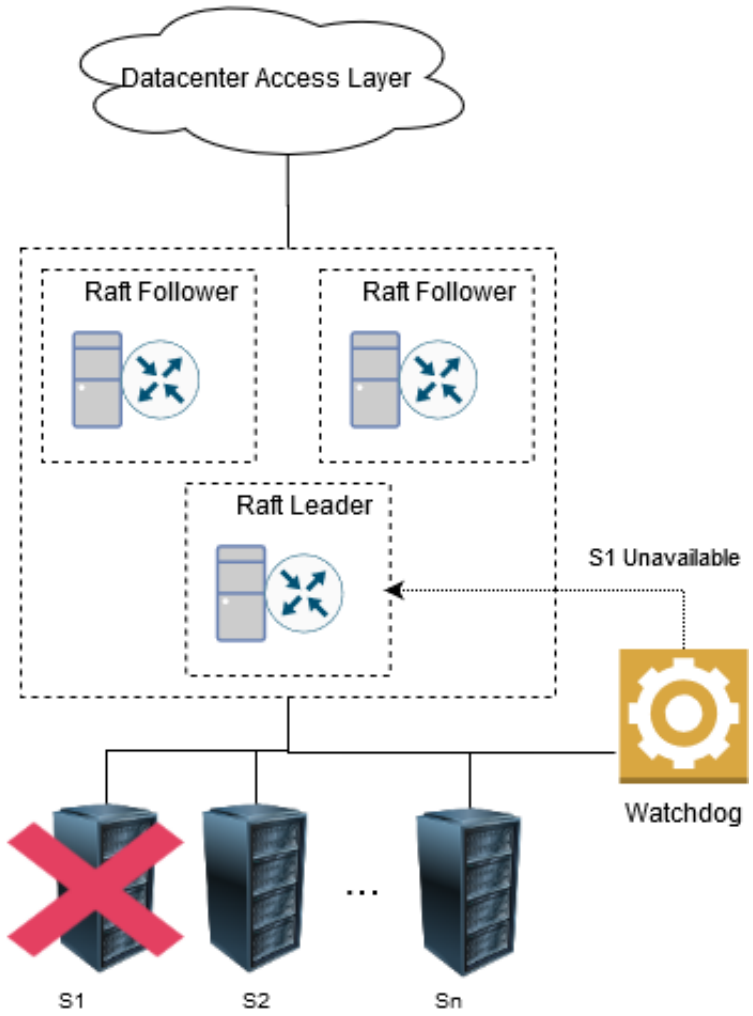


Figure 3.6: The watchdog service notifies the unavailability of S1, triggering the update process within the Raft cluster.

3.3.1.1 Interaction with Raft P4

The adoption of Raft P4 avoids the interaction with external actors needed to perform load balancing. The node itself contains all the information to select the correct output port and forward the packet to its destination server. Figure 3.7 shows the interaction between the watchdog mechanism and the Raft P4, making use of the API. Using an encoding mechanism allows to share the server id and its status within a single request. The encoding uses the first 63-bit to identify the server, and

the last bit to communicate the server status.

Considering the scenario in which the data center is handling web traffic, the procedure to correctly forward an incoming new request is:

- *Calculate the hash.* Calculating $h(r)$, being $h()$ the hash function and r the http request, to identify the position within the hash table.
- *Check the mapped server.* This can be achieved by defining a function that accepts as input parameter the request's hash and returning the assigned server, as shown in Figure 3.5.
- *Forward the packet.* Once the server is correctly mapped, the relative output port is selected and the http request forwarded.

In this scenario, the utilization of Raft P4 can handle the load balancing without relying on external actors, since all the information to forward the packet is saved within the programmable hardware. Assuming server's MBTF (mean time between failures) reasonably high, in case of disruptive events like packet loss while updating the hash table, the time to recover such update is bounded to a single heartbeat.

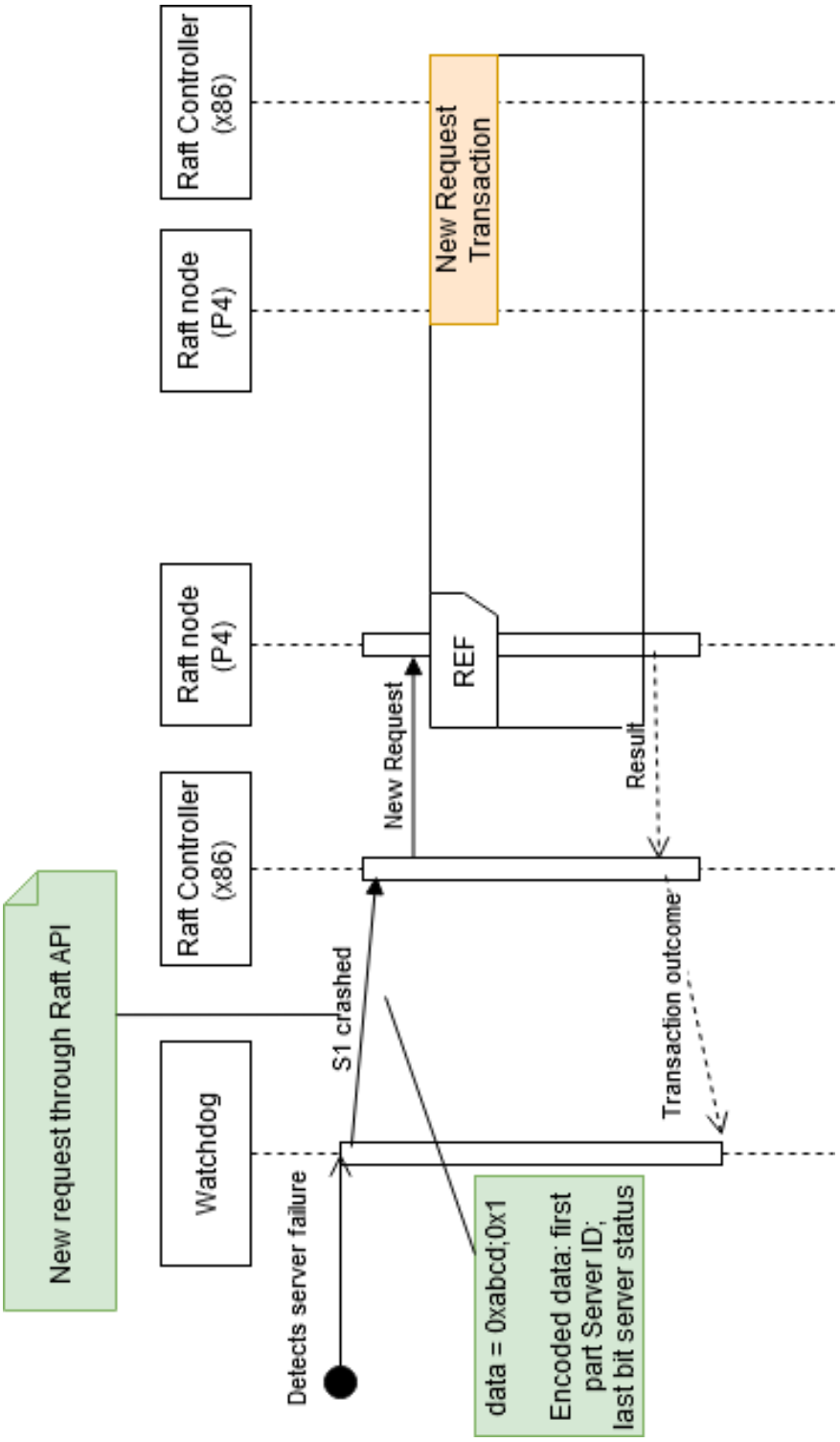


Figure 3.7: The watchdog service notifies the unavailability of S1, triggering the update process within the Raft cluster.

Chapter 4

Architecture and protocol details

4.1 Pipeline

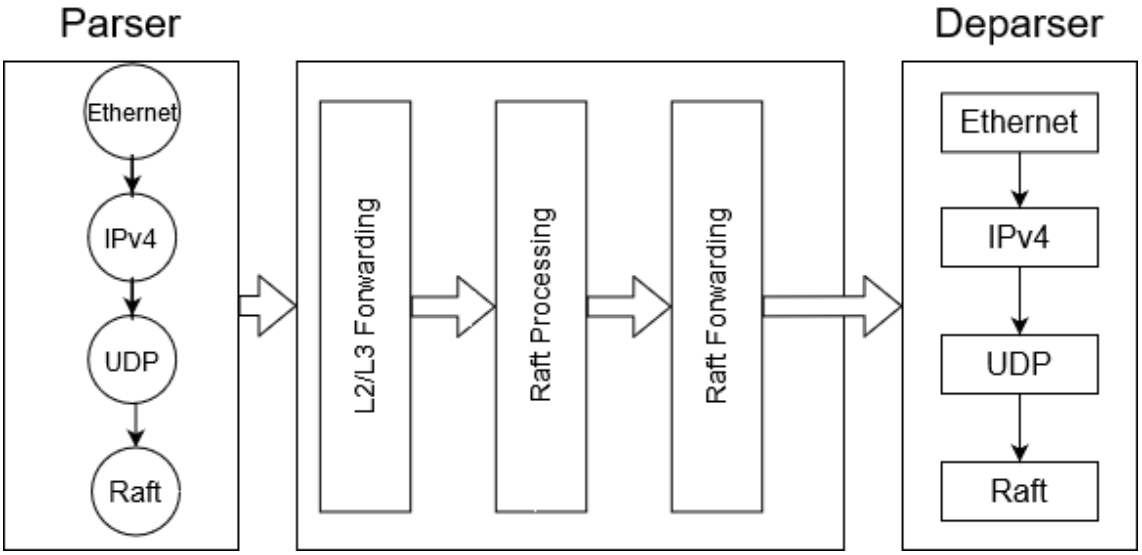


Figure 4.1: Raft P4 Pipeline definition.

This section provides a detailed description on how the pipeline is defined. An high-level representation of how the P4 processing pipeline is designed is shown in Figure 4.1. The *parser* is responsible to output the correct packet header before starting the entire pipeline. Once the parser has finished, the parsed packet will traverse the processing pipeline that will be further analyzed, on a later section. The last step is performed by the *deparser*, that is responsible of 'putting things back together'

while updating the packet checksum before forwarding it to the specified output port.

4.1.1 Registers

Table 4.1: Registers used in P4 application.

Register	Value max bit size	Brief description
logValueRegister	64	corresponds to Raft's log. Used to save values.
logIndexRegister	32	Used to save the actual log size.
currentTermRegister	32	It contains the actual Raft term.
stagedValueRegister	64	Serves as staging area, to complete the transaction.
stagedValueFlagRegister	1	Flag to check if node is in transaction mode.
countLogACKRegister	16	Used to count the actual replies to an Append Entries message.
IDRegister	16	Set by controller plane, it contains the P4 node ID.
majorityRegister	16	Set by controller plane, it contains the cluster's quorum value.
countVoteRegister	16	Used to count the vote replies, in leader election phase.
leaderRegister	16	It contains the ID of the current leader.
roleRegister	2	Used to store persistently the current role of the P4 node.

Storing some information persistently in P4 requires the use of registers, as explained in section 2.1.1. Since P4 does not provide any other option to store any information in persistent mode, we had to make use of registers, even though they introduce a significant overhead in terms of performance, since the access to a register is an atomic operation. Further performance evaluations are left to be analyzed on a later chapter.

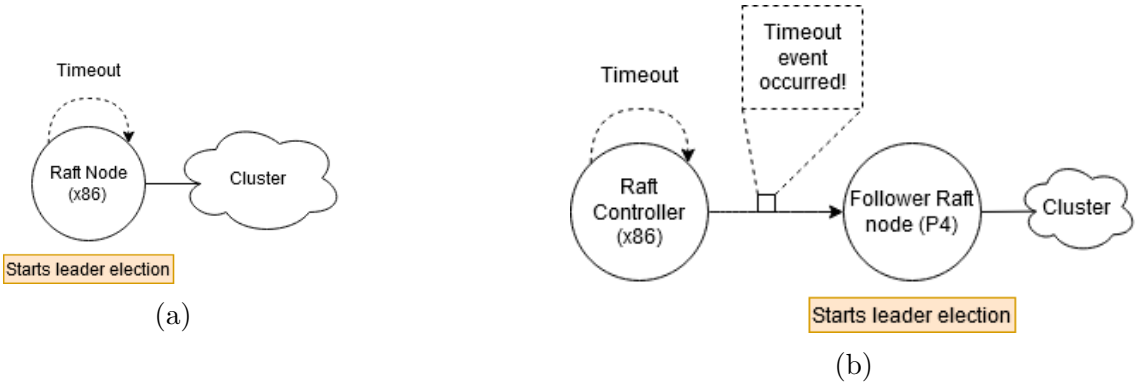


Figure 4.2: Timeout's procedure difference between original Raft (a) and Raft on P4 (b).

Table 4.1 provides a list on the registers that are actually used by the P4 application, along with a brief description on their roles.

4.2 Protocol

This section will further describe the workflow of messages and more details in general about the P4's implementation. Before proceeding, it is important to remark that the original Raft implementation makes use of four kinds of message types that are *heartbeat, append entries, vote request and vote response*, since one of Raft's main goals is to ease human comprehensibility about consensus algorithms, by distinguish every consensus phase as an independent concept. Unfortunately, due to the *match-action* abstraction and the non-Turing complete nature of the P4 language, we had to adopt more message types along with some assumptions, that will be introduced later on, in order to make the application work.
The main differences between protocol's message types is shown in Figure 4.3. Besides message types, to cope with P4's inabil-

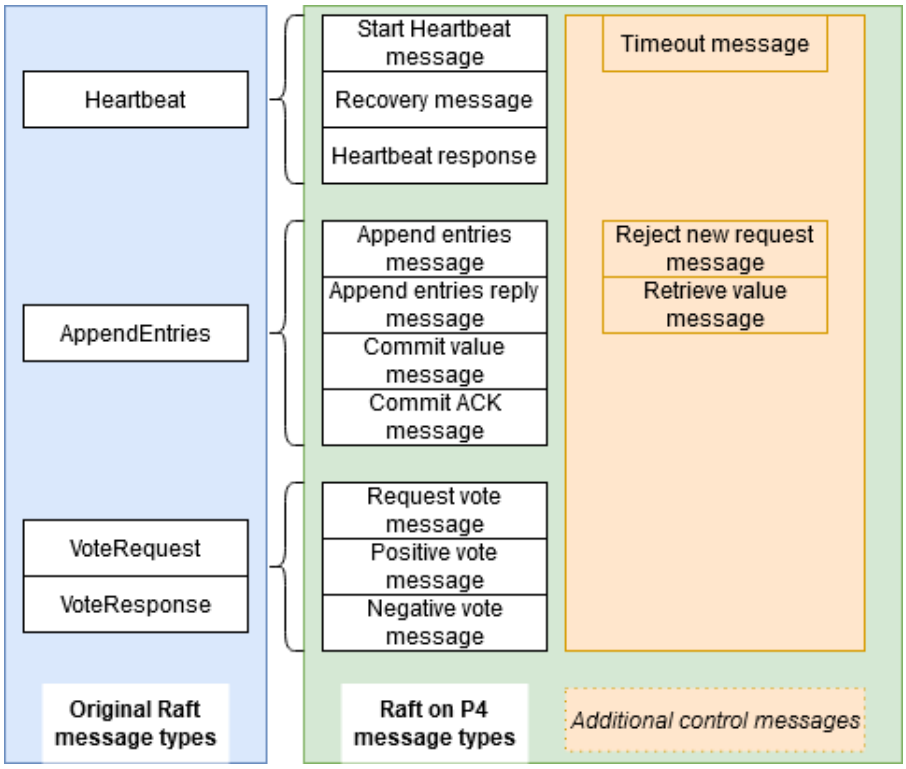


Figure 4.3: Differences between Raft original protocol's messages and Raft on P4 messages.

ity to defines time-based events, we had to redefine the timeout procedure. Referring to Figure 4.2a, the original Raft timeout procedure is handled within the node itself, while in P4 the timeout is reshaped in form of packet-in event, using the relative message type, as shown in Figure 4.2b.

4.2.1 Initialization

The initialization is one of the crucial part of Raft on P4. The figure 4.4 is an illustration about a possible configuration on how to setup the Initialization phase. The *starter agent* can be a service hosted somewhere inside the network, or even a dedicated machine, that is responsible to:

- Execute the *Switch register manager tool* to initialize the P4's registers (e.g. Node ID Register)

- Start the *Raft controller* application using Python.
- Start the http endpoint on port TCP/8080 to expose the entire cluster and provide an access point in case of membership changes about the overall topology[1].

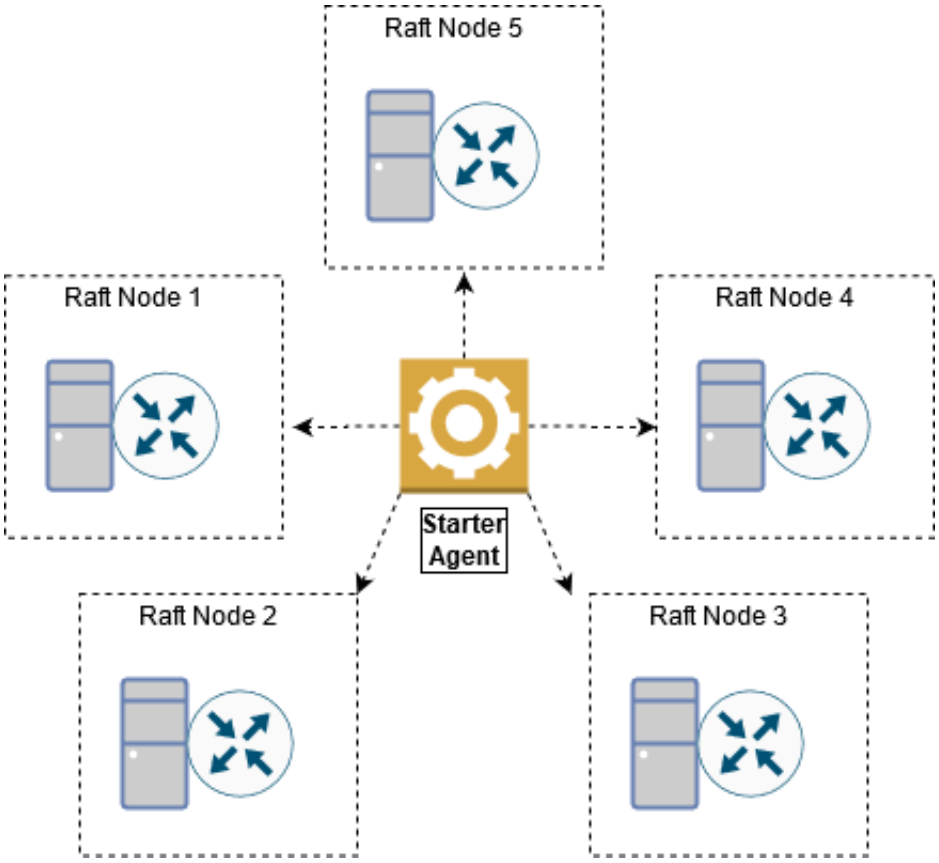


Figure 4.4: Raft on P4 initialization phase.

[1]script *http_endpoint.py*

4.2.2 Leader election

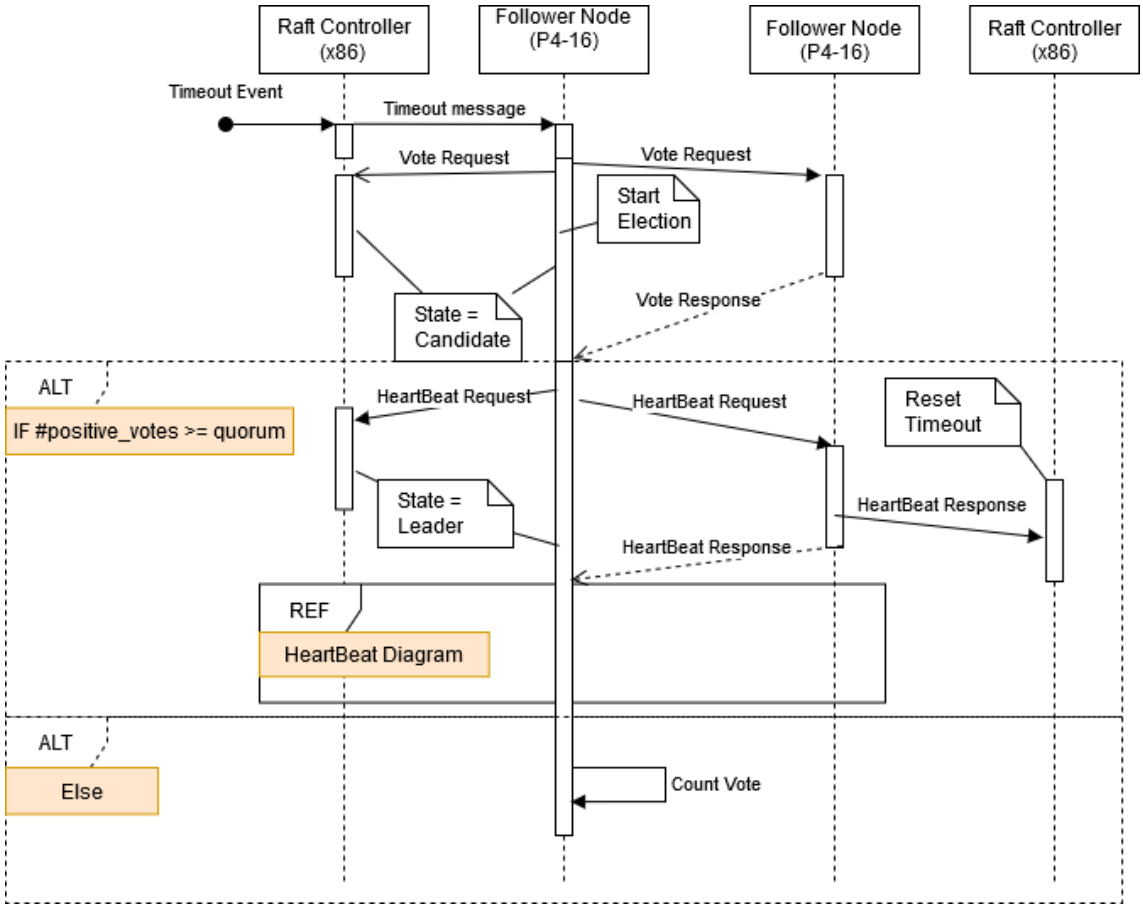


Figure 4.5: Leader Election workflow.

Figure 4.5 illustrates the overall workflow of how Raft P4 handles the leader election. As already described in the previous chapters, all the logic related to timings is delegated to the Raft controller, an application we developed using Python. After the P4 nodes initialization and the execution of the relative controller, the latter will instantiate a thread for the sole purpose of notifying the P4 Node that a timeout event occurred.

Once the timeout event occurs, the state changes to *candidate* first in the Raft controller and then on the relative P4 node by receiving the *timeout* packet. Once the status has been changed, the P4 application will start the leader election

process by sending a *vote request* in multicast.

Upon receiving a *vote request*, a follower node checks whether the requester has enough log entries, along with the term. This logic could not be handled with a *match-action* abstraction, since tables do not provide matching operators like 'greater-than' or 'less-than', this kind of logic had to be specified with conditional statements, within the *control section*.

4.2.3 Heartbeat sequence

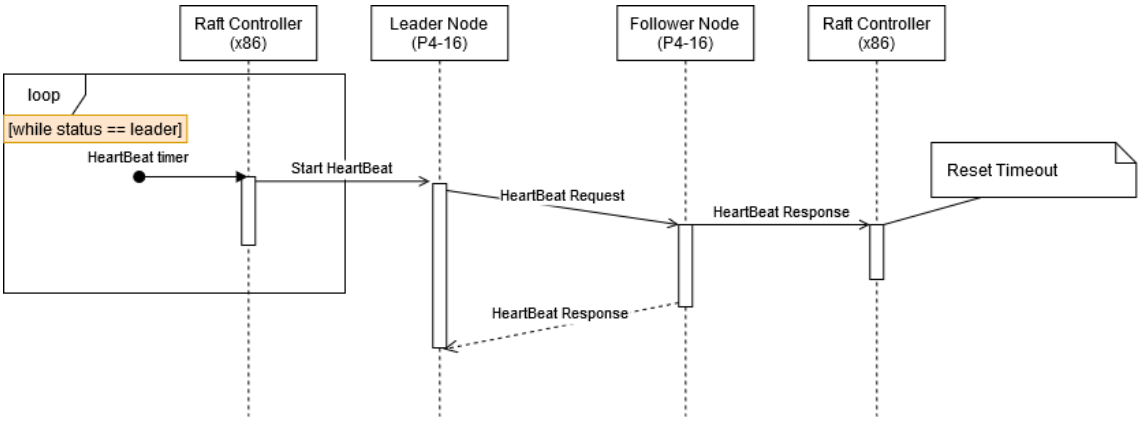


Figure 4.6: Heartbeat workflow.

Once a node has been elected as leader, the associated Raft controller will start a new thread responsible for the crafting of *start heartbeat* Raft packets and sending them to its associated P4 node, hence, starting the heartbeat sequence at P4 layer. The leader node will then spread *heartbeat request* messages with destination ID a multicast value. On the other hand, when a follower P4 node receives a *heartbeat request* from a leader node, it will answer with a unicast Raft packet, cloning it also for its Raft controller since the Heartbeat are also responsible for the reset of the timeout. The overall work-

flow is described, making use of a sequence diagram, in figure 4.6.

4.2.4 Node recovery

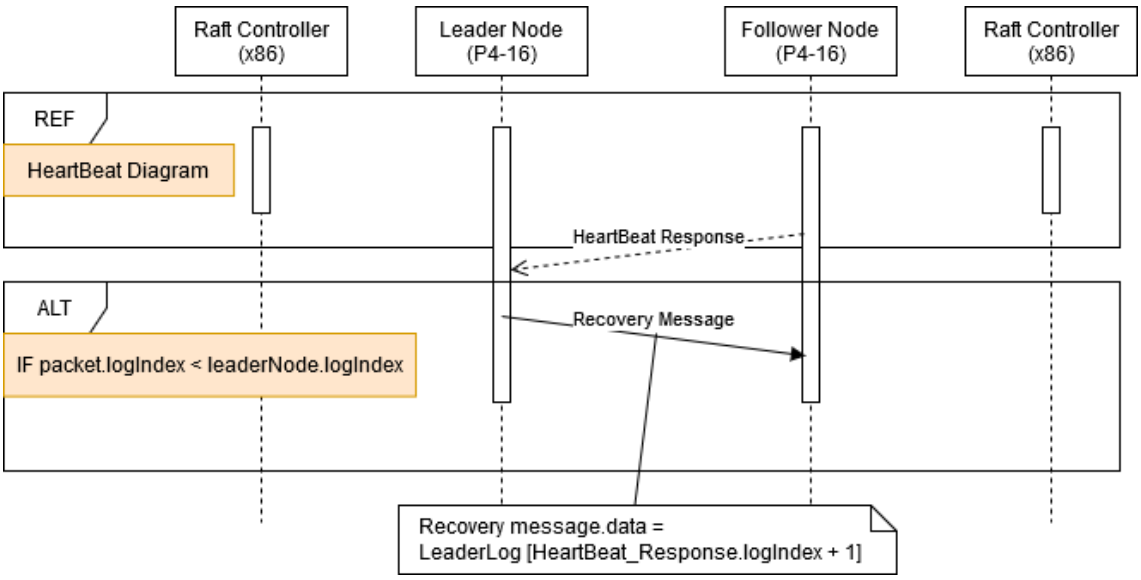


Figure 4.7: Recovery workflow.

In case of node failure, new log entries will inevitably become lost. To overcome this issue, the recovery phase is vital. Referring to figure 4.7, Whenever a follower P4 node will answer to a *heartbeat request*, it will also include the information about its log index. By knowing the follower's log index, the Leader is able to perform the recovery action, i.e. replying to the follower node whose log is smaller than the Leader's, with a recovery message.

4.2.5 New requests

Upon the event of a client sending a *new request* message through the API, depending on the P4 node's state, a new transaction will take place, as already explained in section

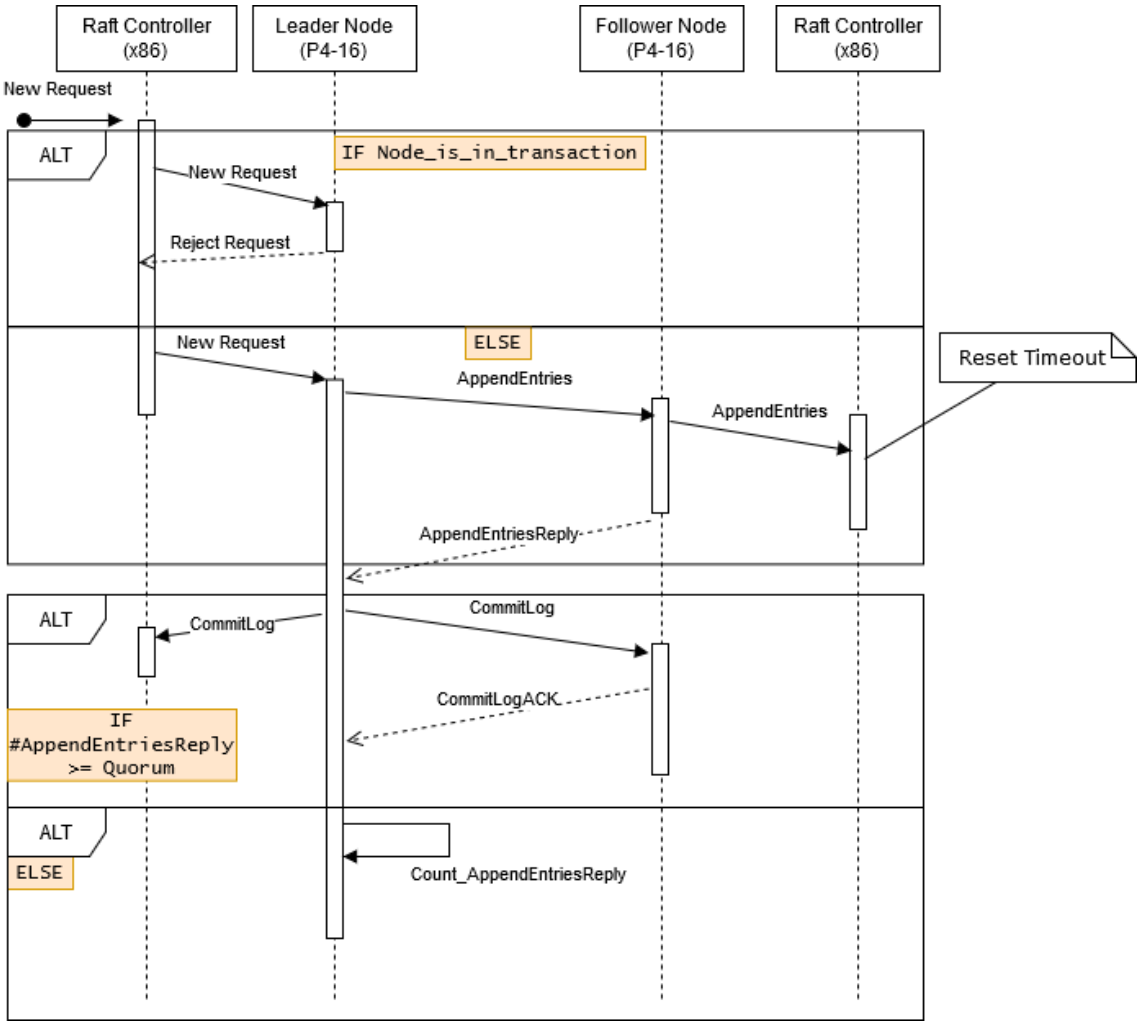


Figure 4.8: New Request workflow.

3.2.3. A representation of the exact workflow of replicating a new value within the cluster is showed in figure 4.8, covering either the successful replication and new value rejection scenarios.

4.2.6 Read log

The retrieval of values that have been replicated is fairly easy: the sequence starts with a *retrieve log request*, with the specified log index to the value to retrieve. The node will return the value, only if the log index specified within the packet does not exceed the number of values stored inside the log.

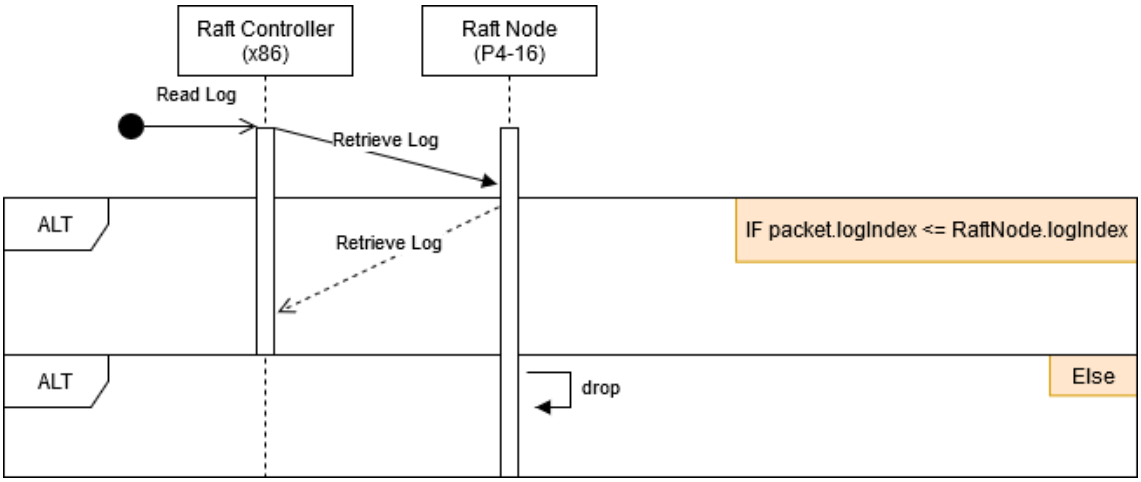


Figure 4.9: Read log workflow.

4.3 Limitations

The protocol has various limitations. Regarding the flexibility, we defined the packet header as TLV fields, as explained in section 3.2.1. As P4, in the actual state of development, does not provide full support about variable length fields, we opted to use fixed length fields.

Concerning the recovery, we developed a 'linear' recovery, meaning that if a node loses several new entries, at most one entry per heartbeat can be recovered. Here, the best approach would be to send a snapshot of the entire log, hence speeding up the recovery mechanism and avoiding the possibility of log inconsistencies. Unfortunately, P4 does not provide any construct to send snapshots instead of packets, thus, preventing us to use the snapshot approach.

Besides the protocol, the P4 application does not support network loops. Since the problem is very well known and resolved by *Spanning-Tree* protocols, we assume that no cycles are present upon the execution of the P4 application. By not

satisfying this condition, broadcast storms are very likely to happen.

4.4 Tables and actions

Tables and actions are the pillars of P4. Table 4.2 represents the structure and content of the P4's leader table, defined within the application. The left side of the table describes the matching values in order to invoke the corresponding action. The asterisk operator (*) denotes the *don' t care* operator.

Match		**Action**
Raft MessageType	Ingress Port	
Heartbeat Request	*	leader_step_down
Append Entries	*	leader_CountCommitACK
Start Heartbeat	Controller Port	leader_spread_heartbeats
New Request	Controller Port	spread_new_request
Heartbeat Response	*	send_to_controller

Table 4.2: Raft P4 Leader Table.

4.4.1 Actions

Using an analogy, actions in P4 are somehow similar to functions, in every object-oriented programming language. Furthermore, it is important to remark that P4's actions do not provide a return value and do not support conditional statements. Actions can be concatenated, in order to achieve a sort of modularity, such that each module contains everything necessary to execute only one aspect of the desired functionality and avoid code repetitions. An example of such modularity is provided in Listing 4.1: *line 9* refers to a previous

defined action whose signature is *multicast()* [2], that provides the functionality of replicating the packet on multiple ports. For brevity, only some of the overall actions, defined within the P4 application, will be listed and described.

```
1    action spread_new_request() {
2
3    stagedValueRegister.write(0, hdr.raft.data);
4    stagedValueFlagRegister.write(0, TRUE);
5
6    hdr.raft.sourceID = meta.raft_metadata.ID;
7    hdr.raft.messageType = APPEND_ENTRIES;
8
9    multicast();
10 }
```

Listing 4.1: *spread_new_request action* definition.

4.5 P4 Control section

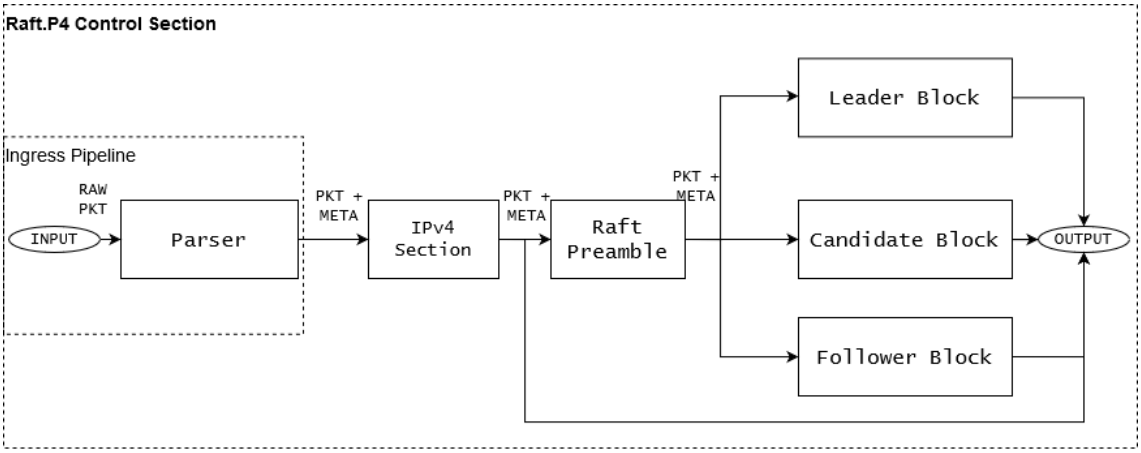


Figure 4.10: P4 application control section

The control section is the P4 application's core. All the algorithm logic that did not suit the *match-action abstraction* is

[2]action defined within Raft.p4 available at https://github.com/EmanueleGallone/RaftP4

described here, in the form of *If* statements. This section will describe in details the logic.

4.5.1 IPV4 section

After the parser has completed its job, the first block processes IPv4. Within this section a IP table is applied to match the packet's IP destination address to identify the correct egress port, in order to forward every packet that has the network layer, supporting, hence, ICMP and other protocols that relies on IP.In case of Raft packet, the egress port previously selected by the IP table match, is later overwritten.

4.5.2 Raft preamble

The Raft preamble function is to provide some preliminary operations. Whenever a new Raft packet enters the pipeline, before applying any logic, the switch needs to retrieve its state by reading all the registers regarding its current role, log index, etc. Moreover, the switch checks if the packets type is write requests or read request from external clients. This way, in case of non-valid packets, we can bypass the entire pipeline and drop them, in order to save resources and increase performance.

Block 1: Raft preamble block pseudo-code.

```
1  RetrieveNodeState() // read all registers to retrieve node state and start
     processing
2
3  if packet.messageType == NewRequest then
4      packet.term ← node.currentTerm // The term is ignored by external clients.
         Updating packet term so that it won't be dropped.
5
6      if node.isInTransactionMode() then
7          packet.messageType ← RejectMessage
8          reply()
9      end
10 end
11 if packet.messageType == RetrieveLog then
12     if packet.logIndex ≤ node.logIndex then
13         packet.data ← node.logValue[packet.logIndex]
14         reply()
15     end
16 end
17 if packet.term <node.currentTerm then
18     drop() // dropping all outdated packets.
19
20     exit() // Exit the pipeline
21
22 end
23 raftForwarding.apply() // custom Raft forwarding, based on node IDs.
24
25 switch node.role do
       // Selecting next Block in control section
26     case Leader do
27         → [Leader Block]
28     end
29     case Candidate do
30         → [Candidate Block]
31     end
32     case Follower do
33         → [Follower Block]
34     end
35 end
```

4.5.3 Leader block

Subsequently to the leader election, the leader node will set its *role register* to the leader value. This way, whenever a new packet is detected in ingress, the Raft preamble will select the leader block. Block 2 represents the *Leader block* in pseudo-code.

Block 2: Leader block pseudo-code.

```
   // node state and metadata loaded in Raft Preamble.
 1 leaderTable.apply() // applying leader table.
 2
 3 if packet.messageType == AppendEntriesReply then
 4 |   read countLogACK register // Counting how many nodes accepted the new value
 5 |
 6 |   if Quorum is reached then
 7 |   |   commit_value() // consolidating the new entry in log in all cluster.
 8 |   |
 9 |   end
10 end
11 if packet.messageType == HeartbeatResponse then
12 |   if packet.logIndex ≤ node.logIndex then
   |   |   // follower node's log is not updated.  starting recovery.
13 |   |   recoveryMessage()
14 |   end
15 end
16 if packet.messageType == VoteRequest then
17 |   spread packet() // Spread the vote request to other nodes, in case of
   |    multi-hop cluster.
18 |
19 |   if packet.logIndex ≥ node.logIndex then
   |   |   // Another node with greater term has started election.  Positive
   |   |    vote since it is updated and his term is greater than mine
20 |   |   step_down()
21 |   |   positive_vote_reply()
22 |   end
23 end
```

4.5.4 Candidate block

The candidate block is responsible to handle the vote requests, mainly. If the P4 node detects valid *heartbeat* or *vote request* messages generated by another cluster member, the node will react by changing its state to *follower*.

Block 3: Candidate block pseudo-code.

```
   // node state and metadata loaded in Raft Preamble.
1  candidateTable.apply() // applying candidate table.
2
3  if packet.messageType == PositiveVote then
4      read countVote register // Counting how many nodes voted for electing node
        as leader.
5
6      if Quorum is reached then
7          roleRegister ← 2 // 2 represents the leader role.
8
9          spread_heartbeat() // The heartbeat informs either the rest of the
            cluster about the correct election and the relative node's Raft
            controller.
10
11     end
12 end
13 → [Output]
```

4.5.5 Follower block

Whenever a leader is elected, the remaining nodes within the cluster will act as follower. This block will handle all the Raft follower's logic. In normal executions, a Raft leader is operating while the rest of the cluster's members behaves as followers. Block 4 will provide a detailed description of the follower block.

Block 4: Follower block pseudo-code.

```
   // node state and metadata loaded in Raft Preamble.
1  followerTable.apply() // applying follower table.
2
3  if packet.messageType == VoteRequest then
4  |   spread packet() // Spread the vote request to other nodes, in case of
   |    multi-hop cluster.
5  |
6  |   if packet.logIndex ≥ node.logIndex then
7  |   |   positive_vote() // voting positively since requester is updated
8  |   |
9  |   else
10 |   |   negative_vote()
11 |   end
12 end
13 → [Output]
```

Chapter 5

Results

5.1 Setup

We used 3 machines with Linux Ubuntu 20.04, installing P4-dev tools using the install script publicly available[1], editing the C/C++ compiler flags of BMV2 for the best performance setup[2].

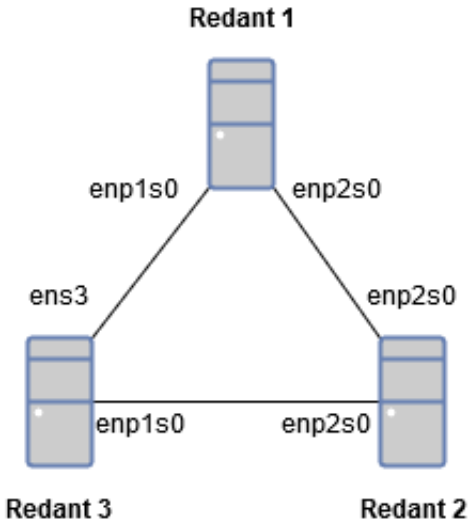


Figure 5.1: Laboratory setup topology listing and relatives Ethernet interfaces. Redant * identifies the machines' hostname.

[1]https://github.com/jafingerhut/p4-guide/blob/master/bin/install-p4dev-v3.sh
[2]https://github.com/p4lang/behavioral-model/blob/master/docs/performance.md

The cluster's hardware specifications are:

- *Redant 1.* **CPU**: Intel(R) Core(TM) i5 660 @ 3.33GHz, **RAM**: 8GB
- *Redant 2.* **CPU**: Intel(R) Core(TM) i5 660 @ 3.33GHz, **RAM**: 8GB.
- *Redant 3.* **CPU**: Intel(R) Core(TM) i5-6500 @ 3.20GHz, **RAM**: 16GB.

Referring to figure 5.1, the machines are connected in a ring topology using *UTP Ethernet* cables running at 1Gbit/s. Upon executing the BMV2 software switches and the Raft P4 application, one link has to be disabled in order to use only one side of the ring, since the current implementation of the application does not support network cycles, as already described in section 4.3.

Since the Raft controller communicates with the collocated P4 software switch through packet-in events, we created a pair of virtual interfaces called *ve_A* and *ve_B*, as represented in Figure 5.2. By doing so, each machine performs both the P4 node and the relative Raft controller work.

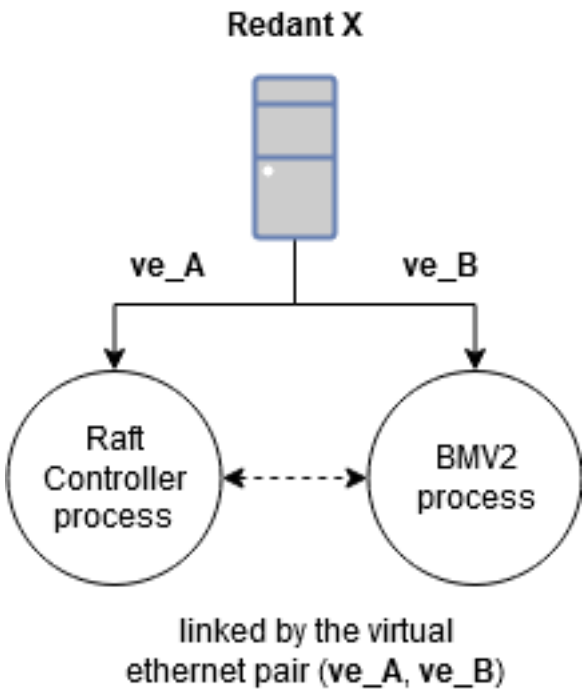


Figure 5.2: Interfacing Raft controller with P4 node through virtual ethernet.

5.2 Tests performed

In this section we are going to describe all the measures and relative setup used to analyze the overall Raft P4 performance.

5.2.1 BMV2 traversal time

To measure the BMV2 traversal time, we used the `TCPDump` tool. By doing so, we were able to compare the differences introduced by the Raft P4 packet processing and simple IP packet processing. The measures were taken by picking the timestamps on the ingress and egress ports, for each packet.

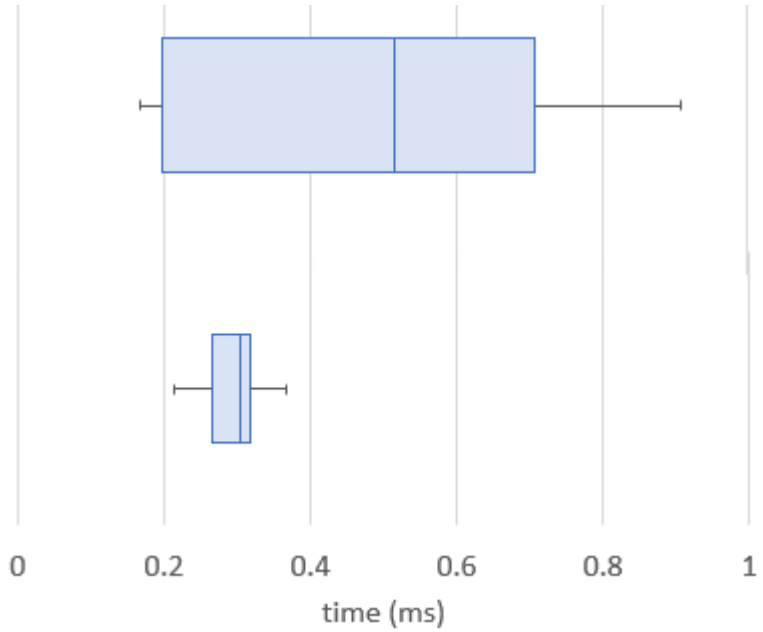


Figure 5.3: The top Whisker box refers to the traversal timings of BMV2 execution with Raft processing, while the bottom refers to BMV2 execution without Raft processing. The reason behind such difference is that before executing any control logic, a sequential reading of all the registers is performed, retrieving the current P4 node's state.

5.2.2 Protocol overhead

By using Raft on P4, some overhead must be taken into account. The overhead, introduced by the protocol, is measured and showed in Figure 5.4 and 5.5. Referring to Figure 5.4, the traffic is relative to the *heartbeat mechanism*, responsible to verify the node's availability. Figure 5.5 represents the total overhead to replicate a new entry within the Raft log. Both plots do not take account of relay messages, responsible to spread the information in case of multi-hop topology.

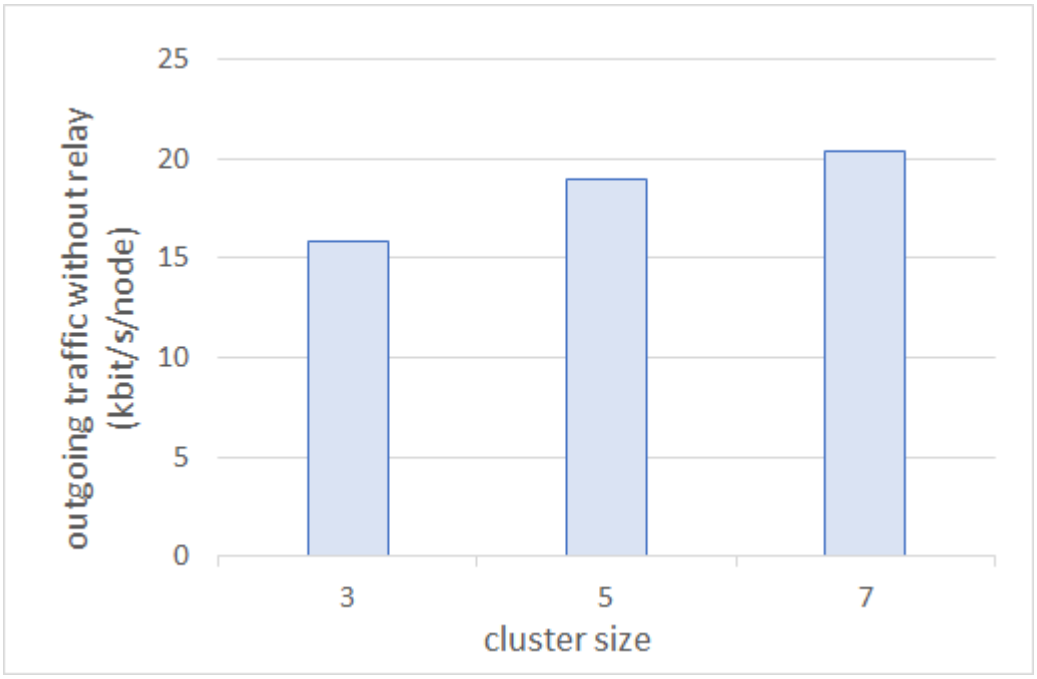


Figure 5.4: Heartbeat mechanism overhead using Raft default timings: one heartbeat every 50 ms.

Figure 5.5: Raft new log entry replication overhead.

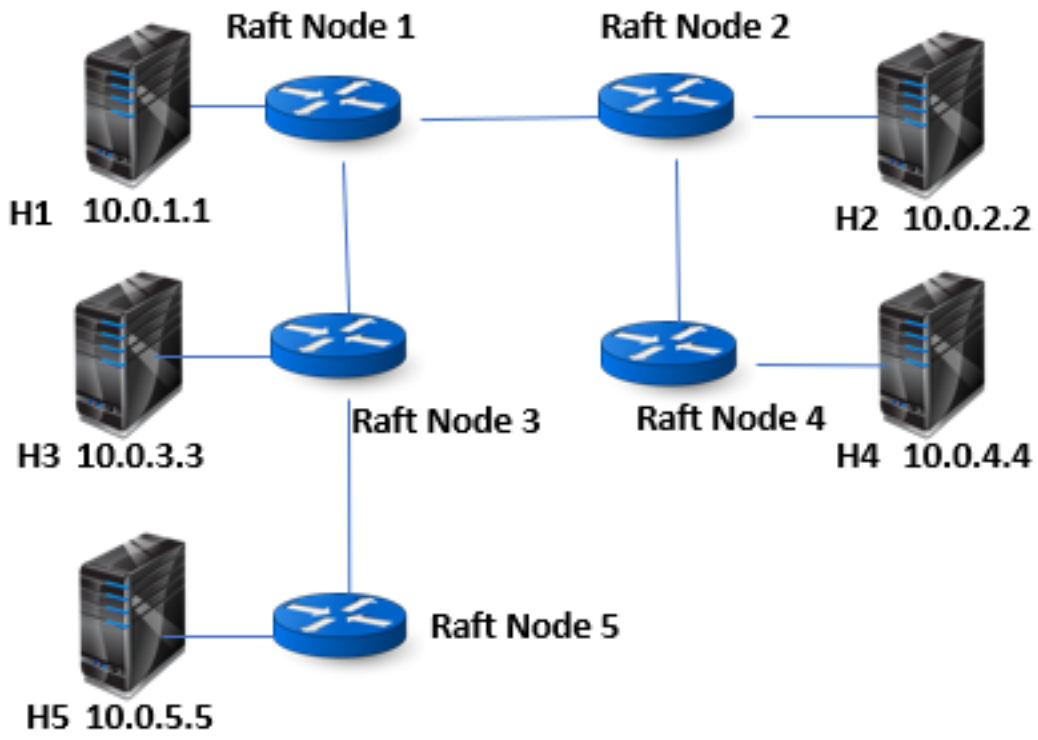


Figure 5.6: Mininet's topology. Each Raft node has a directly linked x86 host executing the Raft controller.

5.3 Mininet setup

To perform measures that required the introduction of link delay, we used the Mininet [39] emulation environment. The topology used is shown in Figure 5.6. The Mininet emulation was executed using *Redant 3*.

5.3.1 Leader election

The leader election timings are strictly correlated to the number of links that a candidate node has to traverse, reaching the quorum of votes and finally change its status to leader, as shown in Figure 5.7. The measurements were taken emulating the topology, shown in figure 5.6, considering *1ms* delay on each link between the P4 nodes. In this particular scenario,

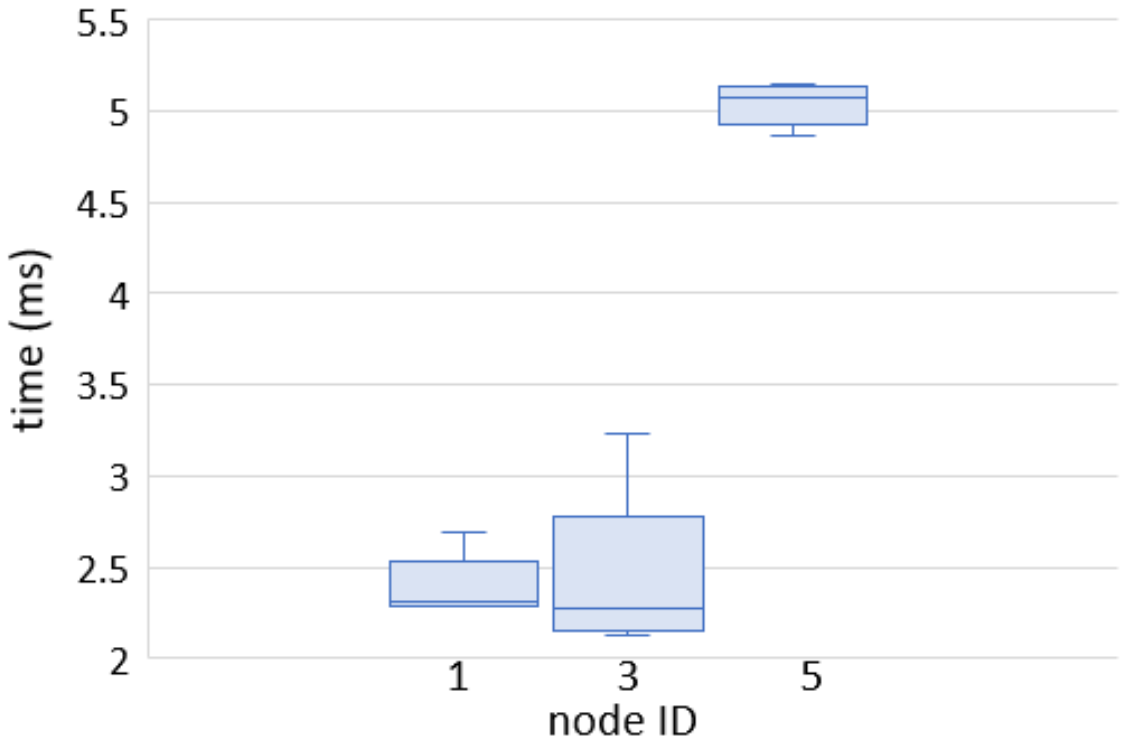


Figure 5.7: Leader election timings. The choice of using Mininet is related to circumvent errors in timing measures, due to clock drift and inaccuracies, in case of measures involving different machines.

the quorum size is equal to three. Taking into consideration the current topology, node 1 has two Raft nodes directly connected to it. Subsequently, the time to reach the quorum is very short. In case of node 5 timeout, the time to reach the quorum increases, since it has only one directly connected Raft node.

5.3.2 Throughput

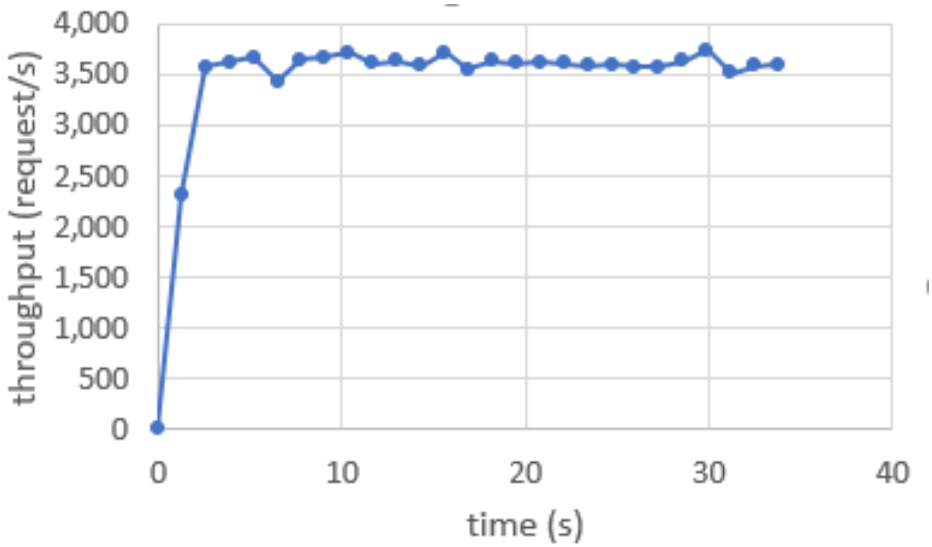


Figure 5.8: Overall Raft P4 throughput in normal operation time.

To measure the application's throughput, we used a topology formed by 3 nodes (deleting nodes 4 and 5, referring to Figure 5.6). In order to both emulate external clients' interaction with the system and measure the overall throughput, we used a custom traffic generator [3] that produces UDP packets which encapsulate the Raft P4's header fields. The measured throughput (represented as requests per second) is shown within figure 5.8. The throughput measurement is evaluated by sampling the P4 software switch at regular time intervals of one second, plotting the three-periods moving average.

5.3.3 Variations in presence of failures

The throughput variation in presence of failures was evaluated to manage such events. Throughput variations are shown in Figures 5.9 and 5.10 in form of two-period moving average

[3] TrafficGeneratorCsharp tool

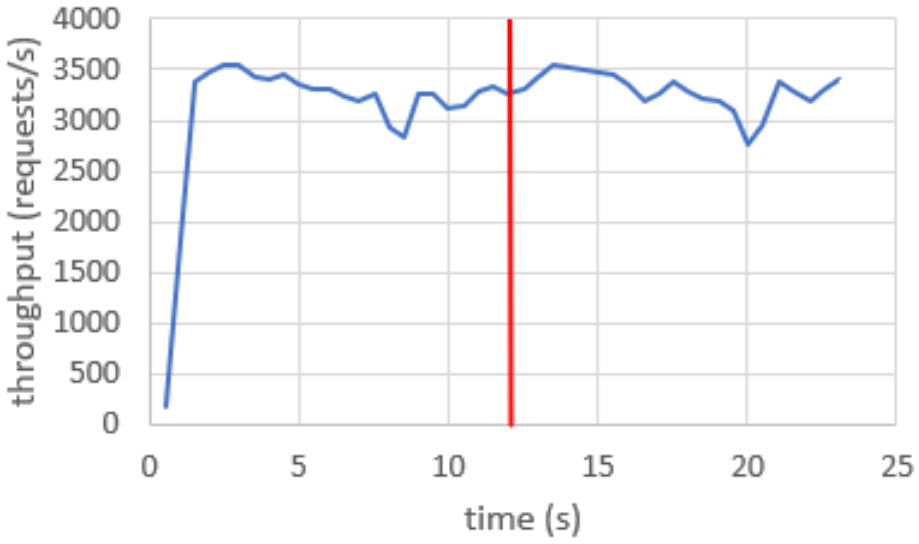


Figure 5.9: Throughput variation in presence of follower failure.

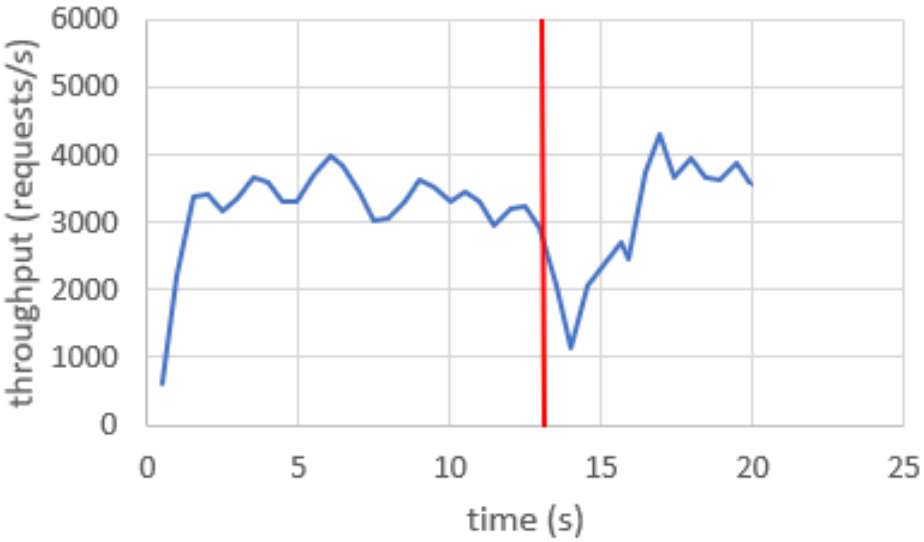


Figure 5.10: Throughput variation in presence of leader failure.

using sampling time = 0.5 s : while a follower's failure does not affect the overall system throughput, a leader node does.

5.3.4 Round-trip time cap

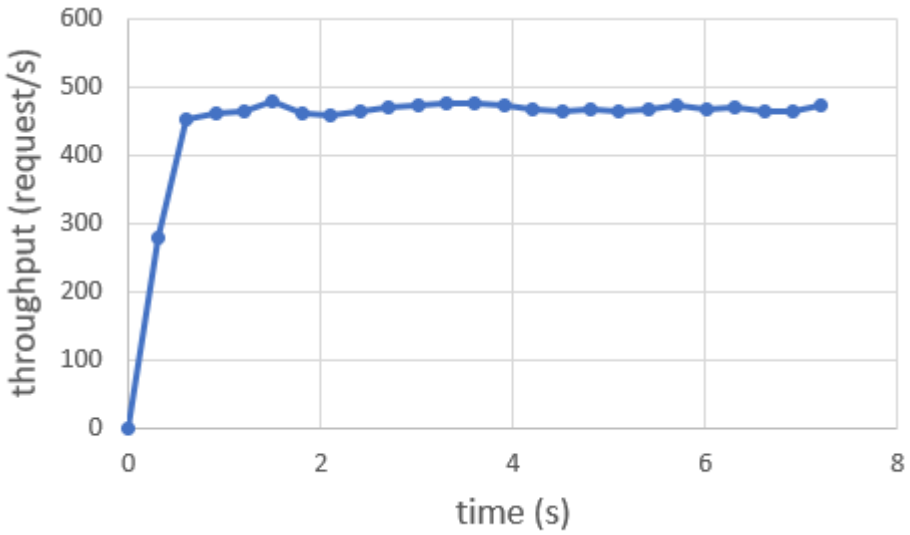


Figure 5.11: Throughput's two-period moving average plot measured within Mininet, with link delay = 1ms. RTT (Round-trip time) = 2ms.

Before committing a new value, the leader node has to check whether the cluster's majority accepts it. As a consequence, the maximum throughput achievable is limited to $\frac{1}{(2RTT)}$, using the protocol's current version. Such limitation is shown within Figure 5.11.

Chapter 6

Conclusions

Consensus is essential to provide high availability and consistency in distributed systems. Consensus algorithms' performance, however, is the main cause of distributed applications' reluctance considering strong consistency. In fact, to cope with the performance degradation, that would be introduced by consensus algorithms, applications are designed to rely on replication mechanisms which offer weak consistency. Although, this approach could lead to data loss in case of particular failure scenarios. Hence, a high-speed consensus service is required. Consequently, improving the overall consensus performance would bring many benefits to distributed applications, especially within data center environments.

While researchers strive to reach higher throughput with low latency on software-based consensus algorithms, a new path has been drawn with the advent of network programmability. We believe that the adoption of expressive data-plane programming languages will have an important impact since it could lead, for example, to new discoveries in terms of network protocols' design, or the possibility of bringing complex application

logic towards the network level.

This work practically showed the possibility to exploit the SDN paradigm by transposing the Raft consensus algorithm from application to network level, adapting it to the *match-action* abstraction that rules the data-plane environment. By offloading the consensus on network level, wire speed timings can be exploited in order to achieve better results in terms of throughput. In addiction, it also shows that network capabilities can be extended, beyond simple connectivity, providing an increased number of services.

Bibliography

[1] Eric Brewer. "A Certain Freedom: Thoughts on the CAP Theorem". In: *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. PODC '10. Zurich, Switzerland: Association for Computing Machinery, 2010, p. 335. ISBN: 9781605588889. DOI: 10.1145/1835698.1835701. URL: https://doi.org/10.1145/1835698.1835701.

[2] Lamport Leslie. "The part-time parliament". In: *ACM Transactions on Computer Systems* 16.2 (1998), pp. 133–169.

[3] Diego Ongaro and John Ousterhout. "In Search of an Understandable Consensus Algorithm". In: USENIX Annual Technical Conference, 2014. URL: https://raft.github.io/raft.pdf.

[4] Dan RK Ports et al. "Designing distributed systems using approximate synchrony in data center networks". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 2015, pp. 43–57.

[5] Zsolt István et al. "Consensus in a Box: Inexpensive Coordination in Hardware". In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, Mar. 2016, pp. 425–438. ISBN: 978-1-931971-29-4. URL: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/istvan.

[6] Murad Kablan. "StatelessNF: a Disaggregated Architecture for Network Functions". PhD thesis. University of Colorado at Boulder, 2017.

[7] Nick McKeown et al. "OpenFlow: Enabling Innovation in Campus Networks". In: *SIGCOMM Comput. Commun. Rev.* 38.2 (Mar. 2008), pp. 69–74. ISSN: 0146-4833. DOI: 10.1145/1355734.1355746.

[8] Weverton Luis da Costa Cordeiro, Jonatas Adilson Marques, and Luciano Paschoal Gaspary. "Data plane programmability beyond openflow: Opportunities and challenges for network and service operations and management". In: *Journal of Network and Systems Management* 25.4 (2017), pp. 784–818.

[9] Tushar Deepak Chandra and Sam Toueg. "Unreliable failure detectors for reliable distributed systems". In: *Journal of the ACM (JACM)* 43.2 (1996), pp. 225–267.

[10] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. "Impossibility of Distributed Consensus with One Faulty Process". In: *J. ACM* 32.2 (Apr. 1985), pp. 374–382. DOI: 10.1145/3149.214121.

[11] *Microsoft Azure*. URL: https://azure.microsoft.com/.

[12] Mike Burrows. "The Chubby lock service for loosely-coupled distributed systems". In: *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2006.

[13] *Apache Zookeeper*. URL: https://zookeeper.apache.org.

[14] Huynh Tu Dang et al. "Paxos Made Switch-y". In: *SIGCOMM Comput. Commun. Rev.* 46.2 (2016), pp. 18–24. ISSN: 0146-4833. DOI: 10.1145/2935634.2935638.

[15] Huynh Tu Dang et al. "NetPaxos: Consensus at Network Speed". In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. SOSR '15. Santa Clara, California: Association for Computing Machinery, 2015. DOI: 10.1145/2774993.2774999.

[16] Marco Primi. *LibPaxos Performance Analysis*. URL: http://libpaxos.sourceforge.net/files/Primim-SPLab08.pdf.

[17] *Libpaxos*. URL: http://libpaxos.sourceforge.net/.

[18] H. T. Dang et al. "P4xos: Consensus as a Network Service". In: *IEEE/ACM Transactions on Networking* 28.4 (2020), pp. 1726–1738.

[19] Leslie Lamport. "Fast paxos". In: *Distributed Computing* 19.2 (2006), pp. 79–103.

[20] *Behavioral model (BMV2)*. URL: https://github.com/p4lang/behavioral-model.

[21] David Hancock and Jacobus Van der Merwe. "Hyper4: Using p4 to virtualize the programmable data plane". In: *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*. 2016, pp. 35–49.

[22] Roberto Bifulco and Gábor Rétvári. "A survey on the programmable data plane: Abstractions, architectures, and open problems". In: *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*. IEEE. 2018, pp. 1–7.

[23] Pat Bosshart et al. "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN". In: *ACM SIGCOMM Computer Communication Review* 43.4 (2013), pp. 99–110.

[24] *OpenVSwtich (OVS)*. URL: https://www.openvswitch.org/.

[25] Han Wang. "Towards a Programmable Dataplane". In: *PhD Dissertation, Cornell University* (2017).

[26] *P4 Documentation*. URL: https://p4.org/p4-spec/docs/P4-16-v1.2.1.html.

[27] *P4 Language Consortium*. URL: https://p4.org/.

[28] *P4Runtime*. URL: https://github.com/p4lang/p4runtime.

[29] *P4C*. URL: https://github.com/p4lang/p4c.

[30] *BMV2 simple switch*. URL: https://github.com/p4lang/behavioral-model/tree/master/targets/simple_switch.

[31] *PSA Architecture*. URL: https://p4.org/p4-spec/docs/PSA.html.

[32] Michael Barborak, Anton Dahbura, and Miroslaw Malek. "The consensus problem in fault-tolerant computing". In: *ACM Computing Surveys (CSur)* 25.2 (1993), pp. 171–220.

[33] Leslie Lamport et al. "Paxos made simple". In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[34] Leslie Lamport. "Lower bounds for asynchronous consensus". In: *Distributed Computing* 19.2 (2006), pp. 104–125.

[35] Miguel Castro, Barbara Liskov, et al. "Practical Byzantine fault tolerance". In: *OSDI*. Vol. 99. 1999. 1999, pp. 173–186.

[36] S. Han et al. "Switch-Centric Byzantine Fault Tolerance Mechanism in Distributed Software Defined Networks". In: *IEEE Communications Letters* 24.10 (2020), pp. 2236–2239.

[37] *Scapy Python library*. URL: https://scapy.net/.

[38] David Karger et al. "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web". In: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. 1997, pp. 654–663.

[39] *Mininet*. URL: http://mininet.org/.