COP: BROAD-SPECTRUM, DEPENDABLE, AND SECURE PROTOCOL ENFORCEMENT MECHANISM FOR MULTI-AGENT SYSTEMS

By

CHEN CONG

A dissertation submitted to the

School of Graduate Studies

Rutgers, The State University of New Jersey

In partial fulfillment of the requirements

For the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Naftaly H. Minsky

And approved by

_____

_____

_____

_____

New Brunswick, New Jersey

May, 2019

ABSTRACT OF THE DISSERTATION

Cop: Broad-Spectrum, Dependable, and Secure Protocol Enforcement Mechanism for Multi-Agent Systems

By CHEN CONG

Dissertation Director:

Naftaly H. Minsky

This work introduces Cop, a highly dependable and secure protocol enforcement mechanism for Multi-Agent Systems. Cop features high scalability, low latency, and regulated interoperability, making it applicable to a broad spectrum of systems including large-scale and complex systems, time-critical systems, and systems-of-systems. Cop enforces protocols via Law-Governed Interaction, coupled with a new protective mechanism that significantly enhances the dependability and security of the enforcement in a highly scalable and efficient manner. Cop is arguably superior to the currently popular Blockchain-based Smart Contract mechanisms in terms of its applicability to a wide range of systems, as the latter is inherently unscalable, has high latency, and lacks interoperability.

Acknowledgments

First of all, I want to thank my advisor Dr. Naftaly Minsky for his guidance, inspiration, encouragement, and support. To me, he is not only an erudite and strict advisor, but also an energetic and amiable friend. Under his guidance, I learned how to think critically, systematically, rigorously, and precisely, as a serious researcher should do. I'll never forget our afternoon coffee chats about computer science, physics, history, philosophy, and art, as well as all those wisdom-filled stories.

I'm very grateful to my dissertation committee: Dr. Naftaly Minsky, Dr. Badri Nath, Dr. Wade Trappe, and Dr. Victoria Ungureanu, for their help and insightful comments to my work. I really enjoyed our discussions.

I'm also very grateful to my friends Zhe Wang and Yue Zhu, Biao Li and Qingxiang Han, Yang Song and Yi Wang, and Liu Liu, for their warm friendship and unconditional help during all these years.

Finally, I'd like to thank my family. I thank my parents for their constant love, support, and understanding, for fostering my deep interest in computer science, and for setting a role model for me with their respectable and scholarly personalities. I thank my wife Mingzhu for her love and camaraderie since high school, for her company through this exciting but arduous adventure, for her happiness for every progress I made, and for her witness of my sweat and tear for this work; it's my greatest fortune to be able to share with her all the sunshine, storms, and rainbows.

Dedication

To my family.

Table of Contents

Chapter 1

Introduction

Multi-Agent System (MAS) is a type of distributed system consisting of a group of loosely coupled, autonomous, and heterogeneous actors interacting with each other to solve problems beyond individual capabilities. The actors in MAS may be software processes, physical devices, or people. A group of actors in a MAS, who may not trust each other, may be required to interact with each other subject to a given interactive protocol, we call such a group a *community*.

Some protocols can be established by the voluntary compliance of all actors in the community. As pointed out by [22], such a protocol must satisfy the following conditions:

- Complying with the protocol is in the vested interest of all actors.
- Failure to comply with the protocol by any actor should not cause serious harm to other actors.

If any one of these conditions is not satisfied, then the protocol needs to be enforced. We call a mechanism used for enforcing protocols a *Protocol Enforcement Mechanism* (PEM). We maintain that an effective PEM should have the following qualities:

- It should be *Dependable and Secure*, i.e., able to defend itself against failures and attacks.
- It should be *Broad-spectrum*, i.e., able to support a wide range of systems including time-critical systems, large-scale systems, complex systems, and systems-of-systems. This leads to the following further requirements:
  - *Sufficiently Low Latency.* Latency means the time it takes the PEM to resolve a transaction. Different applications may require different upper bounds of latency.

For commercial and financial systems it may be in the magnitude of minutes, while in time-critical systems such as industry control systems and airborne control systems, it may be in the magnitude of milliseconds.

- *High Scalability.* Scalability means the ability to accommodate the increase of transaction volume and frequency without significantly increasing the latency. Scalability is crucial to large-scale systems, such as enterprise systems, infrastructure systems, and financial systems.
- *Regulated Interoperability.* Interoperability means the ability of communities that operate subject to different protocols to interact with each other, and such an ability needs to be subject to regulations on which communities can interact with each other and how such interactions should be conducted. Regulated interoperability is required by complex systems and systems-of-systems such as federated enterprise systems, supply chain systems, and healthcare systems, in which different systems need to interact with each other while each of them operates under its own protocol. Moreover, most complex systems and systems-of-systems cannot be effectively governed by a single protocol, they generally require different parts of the system to operation subject to different protocols, while still being able to interact with each other.

Recently, a type of PEM for MAS inspired by [54] and [43] called *Blockchain-based Smart Contract* has been attracting much attention from both academia and industry. Some most widely known Blockchain-based Smart Contract implementations include Ethereum [58] and Hyperledger Fabric [1]. Blockchain-based Smart Contract records transactions in a temper-free ledger called *blockchain* maintained within a distributed network of mutually untrusting peers and can enforce protocols over the blockchain through the underlying consensus among the peers. It features a high level of dependability and security; however, it lacks the quality of broad-spectrum due to the following inherent limitations:

- The consensus procedure of Blockchain-based Smart Contract leads to substantial latency, including (a) proposal latency: the time it takes a transaction to be included in a block proposal; (b) broadcasting latency: the time it takes a block proposal to

be broadcast to all peers; (c) ordering latency: the time it takes for different block proposals to be ordered; (d) validation latency: the time it takes a block proposal to be validated and admitted by all peers. Under the various implementations of Blockchain-based Smart Contract, the latency can be in the magnitude of minutes or more.

- Blockchain-based Smart Contract is inherently unscalable, even if many works such as [1] tried to reduce the level of unscalability. Although the consensus has a decentralized nature, the enforcement itself is essentially centralized and linear: a new block cannot be admitted to the distributed body of the blockchain until its preceding block is resolved via the decentralized consensus. Suppose each block contains b transactions in average, and it takes $t(b)$ time to resolve one such block in average, to resolve s amount of transactions, it requires $s \times t(b)/b$ time. With the increase of transaction volume and frequency, the latency will unavoidably grow.

- Blockchain-based Smart Contract cannot effectively handle interoperations between communities, let alone regulated interoperations. Some works including [55] and [27] tried to solve this problem; however, none of them provides a practical and general solution. In fact, this is widely regarded as the "holy grail" of this field.

On the other hand, another PEM for MAS with a much longer history called *Law-Governed Interactions* (LGI), which will be introduced in Chapter 2, satisfies most of our required qualities of an effective PEM, including low latency, high scalability, and regulated interoperability. However, as Chapter 3 will show, LGI has a serious flaw that may degrade its dependability and security, to which we refer as its Achilles' Heel. Fortunately, unlike the inherent limitations of Blockchain-based Smart Contract which are hard or even unable to get rid of, the Achilles' Heel of LGI can be largely removed.

The goal of this work is to create an LGI-based PEM for MAS which satisfies all our required qualities by removing the Achilles' Heel of LGI. This PEM is named *Cop*, aiming to evoke an analogy to police officers who enforce social laws.

The rest of the dissertation is organized as follows: Chapter 2 provides an overview of LGI. Chapter 3 discusses the Achilles' Heel of LGI. Chapter 4 introduces the architecture

of Cop. Chapter 5 and Chapter 6 introduce how Cop removes the Achilles' Heel of LGI. Chapter 7 discusses Cop's requirements on LGI's implementation and our solutions. Chapter 8 discusses a functionally complete prototype of Cop and the experiments we have done with it. Chapter 9 proposes several future research directions. Chapter 10 concludes this dissertation.

Chapter 2

An Overview of LGI

Law-Governed Interaction (LGI) [41] is a decentralized, stateful, and modular mechanism for a community of distributed heterogeneous actors to interact under an explicit and enforced protocol called the *law* of the community. As shown in [60] and [40], LGI can effectively enforce protocols in MAS. This chapter provides an overview of LGI, while its manual [39] provides a more in-depth description.

The gist of LGI is as follows. A community following the law $\mathcal{L}$ is called $\mathcal{L}$-community, denoted as $C_{\mathcal{L}}$. LGI can enforce $\mathcal{L}$ in $C_{\mathcal{L}}$ in a strictly decentralized way without needing the knowledge or control of the internals of the actors in $C_{\mathcal{L}}$, which is roughly done as follows. An actor can join $C_{\mathcal{L}}$ by adopting a private surrogate called *controller* loaded with $\mathcal{L}$ and conducting interactions through this controller, then the actor along with its controller is called an $\mathcal{L}$-agent. The controller conducts interactions strictly following $\mathcal{L}$, and thus enforces $\mathcal{L}$ on the interactive activities of the $\mathcal{L}$-agent. Thanks to the local nature of laws, which will be elaborated in Section 2.1, such enforcement is carried out locally at the $\mathcal{L}$-agent, without needing the knowledge of or dependency on anything that happens simultaneously at other $\mathcal{L}$-agents, enabling $\mathcal{L}$ to be enforced in parallel in $C_{\mathcal{L}}$ by the controllers of various $\mathcal{L}$-agents.

The decentralized nature of the enforcement ensures its inherently low latency and high scalability; in particular, as shown in [41], the latency of LGI is in the magnitude of milliseconds. In addition, as Section 2.3 will discuss, LGI enables two different communities to interoperate under the regulation of both communities' laws. For more complex systems, LGI provides a more flexible mechanism called *Conformance Hierarchy* to regulate the interactions in them.

The rest of this chapter introduces several aspects of LGI. Section 2.1 introduces the

concept of law; Section 2.2 discusses the enforcement of law; Section 2.3 introduces how LGI enables regulated interoperations between communities.

2.1 The Concept of Law

In LGI, a law is formulated in terms of three elements defined with respect to a given controller t:

- E: the set of all *Regulated Events* that may occur locally at t, such as the arrival of a message. Every regulated event may carry multiple arguments and can be uniquely represented by a byte string.
- S: the set of all possible local *Control States* of t. Distinct from the internal state of the actor, of which the law is oblivious, the control state of t is an unbounded but usually small set of key-value pairs, each of which is called a *control state entry*.
- O: the set of all *Primitive Operations* that can be mandated by a law for t to execute, such as sending a message. Every primitive operation may include multiple arguments and can be uniquely represented by a byte string.

The formal definition of a law $\mathcal{L}$ is a mapping:

$$\mathcal{L} : E \times S \to O^* \times S$$

That is, in response to the occurrence of a regulated event at t, according to the control state of t at the time of occurrence of the event, $\mathcal{L}$ mandates a possibly empty sequence of primitive operations that t should execute sequentially and atomically, and a new control state that is to replace the current control state of t.

The definition of law reveals its *local nature*: when making rulings, a law $\mathcal{L}$ only concerns the events that occur locally at the controller and the local control state of the controller, and the operations it mandates are to be executed locally by the controller. Although some operations $\mathcal{L}$ mandates may incur non-local effect (e.g., sending a message to another controller which will trigger an event on that controller), the decision to execute such

operations are still made locally. Despite the local nature, $\mathcal{L}$ has a *global sway* in $\mathcal{L}$-community, as it is locally enforced at every $\mathcal{L}$-agent. Meanwhile, as proved in [39], the local nature does not reduce law's expressive power.

Law is stateful, i.e., the rulings of a law may depend on the controller's current control state, while the evolution of the controller's control state is regulated by the law by mandating *set* operations, which create or update control state entries; and *unset* operations, which remove control state entries. The statefulness of law makes it sensitive to the history of interactions, which is crucial for establishing coordination protocols between them.

A law can be made proactive to an extent with the mechanism of *Enforced Obligation*: it can mandate an *imposeObligation* or *repealObligation* operation for a controller to impose or repeal an obligation, which is essentially a named timer. When the obligation dues, i.e., the specified time is up, an *obligationDue* event will occur at the controller for which the law can make further rulings. Enforced Obligation enables the law to express both safety and liveliness rules.

A law may be non-deterministic, i.e., its rulings may base on random numbers. Non-deterministic laws are specifically useful for the protocols involving randomness and tie-breaking.

A law can be expressed with Event-Condition-Action (ECA) blocks, any language that can express ECA blocks can potentially be developed into a Domain-Specific Language (DSL) for writing laws. In this dissertation, all laws are written in a DSL called *LawScript* which is designed by the dissertation author and is fully supported in the latest implementation of LGI. Appendix A will introduce the details of LawScript.

Law 2.1 and Law 2.2 are two simple example laws written in LawScript. In Law 2.1 (`money` law), each member of the community is provided with an initial budget of $1000 when joining the community, then it can transfer any amount of money to another member only if the amount is not greater than its current budget, after which the budgets of the sender and the receiver will be updated appropriately. Law 2.2 (`monitor` law) establishes a systematic monitoring scheme with a designated monitor for all interactions within the community, so that whenever a new member joins the community or a member sends a message to another member, the monitor is guaranteed to be informed.

Law 2.1: money law

```
Name: money
LawScript: CoffeeScript

UPON "adopted", ->
  DO "set", key: "budget", value: 1000
  return true

UPON "sent", ->
  if @message <= CS("budget")
    DO "set", key: "budget", value: CS("budget") - @message
    DO "forward"
    return true

UPON "arrived", ->
  DO "set", key: "budget", value: CS("budget") + @message
  DO "deliver"
  return true
```

Law 2.2: monitor law

```
Name: monitor
LawScript: CoffeeScript

monitor = "monitor@192.168.1.100:5000"

UPON "adopted", ->
  DO "forward", receiver: monitor, message:
    type: "adopted", controller: @self
  return true

UPON "sent", ->
  DO "forward", receiver: monitor, message:
    type: "sent", message:@message
    sender: @self, receiver: @receiver
  DO "forward"
  return true

UPON "adopted", ->
  if @self is monitor
    return true

UPON "arrived", ->
  DO "deliver"
  return true
```

2.2 The Enforcement of Law

In LGI, laws are enforced by mutually independent controllers in a strictly decentralized way, making LGI free from single point of attack, single point of failure, and low scalability.

All generic controllers are supposed to be maintained by a highly reputable organization called Controller Service (CoS), whose users may trust it just like they trust email services, ISPs, or cloud computing services. CoS can vouch for the authenticity of the generic controllers it maintains so that each of them can be trusted to correctly enforce any well-formed law loaded into it. CoS may ensure the authenticity of controllers with trusted computing technologies such as Trusted Platform Module (TPM) [5], which is particularly feasible to deploy in CoS as all the generic controllers have the identical code which leads to a single and stable hash. Moreover, CoS provides each controller it maintains with a certificate with its signature, so that the controller can authenticate itself as an authentic controller when it interacts with other controllers.

Any actor can join $\mathcal{L}$-community and thus form an $\mathcal{L}$-agent by performing the following steps:

1. The actor acquires a generic controller from CoS.
2. The actor loads the controller with $\mathcal{L}$ retrieved from a well-known and trusted *law server*.
3. The actor adopts this controller to be its surrogate that mediates all its interactive activities subject to $\mathcal{L}$.

The motivation for an actor to join $\mathcal{L}$-community is due to the *handshake mechanism* of LGI: before conducting interactions, a pair of controllers first validate each other's certificate signed by CoS to establish mutual trust on each other's code, then exchange the hash of their laws. In this way, the controllers can mutually recognize each other's law, and can justifiably trust each other to enforce the law. The handshake mechanism enables a law to set its community's boundary of interactions. By default, only interactions within the community are allowed, while communities may also interoperate to the extent permitted

by the respective laws. Therefore, if an actor's intended interlocutor requires it to join $\mathcal{L}$-community, then it is effectively compelled to join $\mathcal{L}$-community.

The adoption of a controller triggers an *adopted* event on the controller, for which $\mathcal{L}$ may make rulings, such as initializing the control state. $\mathcal{L}$ may also reject the adoption under some circumstances by mandating a *quit* operation (e.g., when the actor does not provide a required certificate), which enables $\mathcal{L}$ to set requirements on the actors that are allowed to join $\mathcal{L}$-community.

The interactive activities of an $\mathcal{L}$-agent are governed by $\mathcal{L}$ as follows:

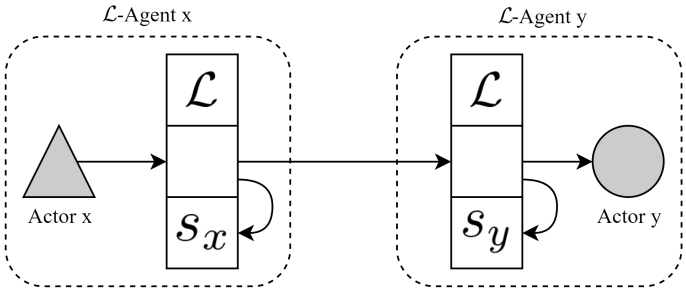
- When the actor of an $\mathcal{L}$-agent wants to send a message to another $\mathcal{L}$-agent, it requests the controller to carry out the interaction, which will trigger a *sent* event on the controller, for which $\mathcal{L}$ can make rulings on what the controller should do.
- To make a controller send a message to another controller, $\mathcal{L}$ can mandate a $forward$ operation.
- When the controller of an $\mathcal{L}$-agent receives a message from another controller, an *arrived* event will occur at the controller, for which $\mathcal{L}$ can make rulings on what the controller should do.
- To make a controller deliver a message to the actor that adopted it, $\mathcal{L}$ can mandate a *deliver* operation.

Figure 2.1 shows an interaction between two $\mathcal{L}$-agents governed by $\mathcal{L}$. Note the dual nature of the governance: the interaction is first mediated by the sender's controller, and then by the receiver's controller.

Sometimes a controller may need to interact with actors not belonging to any community; such an actor is called an *alien*. When the controller receives a message from an alien, a *submitted* event will occur at it. On the other hand, $\mathcal{L}$ can mandate a *release* operation under some circumstances which is executed by the controller by sending a message to an alien.

Note that the actor and the controller of an $\mathcal{L}$-agent may be spatially separated, their

Figure 2.1: An Interaction Governed by $\mathcal{L}$



communication is via the network connection between them. Once the connection is broken, a *disconnected* event will occur at the controller; once the connection is restored, a *reconnected* event will occur at the controller. This allows $\mathcal{L}$ to handle the changes on the connection status.

Sometimes the controller of an $\mathcal{L}$-agent may fail to execute an operation, e.g., when it tries to send a message while the receiver is offline. When such a failure happens, an *exception* event will occur at the controller with the information of the failed operation, then $\mathcal{L}$ can make rulings for this event to handle the exception.

It is noteworthy that every event carries an argument indicating the timestamp when it occurs at the controller, which may affect the rulings of the law. Note that this timestamp does not always reflect the time when the controller processes the event, as an event is not always immediately processed by the controller when it occurs, because the sequence of operations a law mandates are to be executed sequentially and atomically by the controller before processing the next event, i.e., during the execution of this sequence of operations, no other operations should be executed, and no other events should be processed. Therefore, when an event occurs at a controller, it may need to wait to be processed until all the operations mandated for the previous event have been executed.

2.3 Beyond Single Community

LGI can support more than a single community, as the generic controllers maintained by CoS may serve multiple different communities after being loaded with different laws. Obviously, LGI can support multiple isolated communities.

In addition, it can also support multiple interoperating communities. As mentioned above, a law can set its community's boundary of interactions, which is done by specifying the communities its members can interact with, the conditions under which such interactions are allowed, and how such interactions should be carried out. Therefore, members from different communities may interact under the regulation of their respective laws.

Furthermore, LGI can organize a set of laws that collectively governs a complex system or a system-of-systems into a coherent ensemble called Conformance Hierarchy, which is a tree of laws rooted by a single root law, in which every law, except the root law, transitively conforms to its superior law, and this conformance relation is inherent in the way it is constructed, which requires no extra validation. Each law in a Conformance Hierarchy may be used to enforce protocols in a subsystem; when interacting, agents operating under the different laws in a common Conformance Hierarchy, i.e., agents from different subsystems, can identify the position of each other's laws within this Conformance Hierarchy, and thus their interactions can be regulated by their respective laws accordingly. The formal definition and the construction of Conformance Hierarchy are introduced in [2], and a recent application of Conformance Hierarchy is introduced in [38].

Chapter 3

The Achilles' Heel of LGI

The correctness of LGI's enforcement of laws depends on the correct operation of controllers. Even if CoS does its utmost to maintain and protect the authentic controllers, there are still chances that some individual controllers may malfunction and violate the law subject to which they are supposed to operate in the following ways:

- Not executing some operations that their laws mandate.
- Executing some operations that their laws do not mandate.
- Executing different sequences of operations than what their laws mandate.

Such a malfunction is essentially a failure of a controller, which may happen under the following threat model:

- An insider of CoS or an outsider who discovers some vulnerability in the controller's code may conduct a targeted attack on the controller and cause the failure.
- A bug in the controller's code may appear and cause the failure, which can only appear under some rare runtime conditions.

Unlike the case in centralized reference monitors, in LGI, a community as a whole may continue to operate despite the failure of individual controllers. However, individual failed controllers may be able to inflict damage to their community; in some cases, even a single failed controller may be devastating to the entire community. A case in point is Law 2.1 (`money` law): one failed controller may distribute a large amount of fake money among other members of the community without raising any suspicion. This reveals a serious flaw of LGI to which we refer as its Achilles' Heel: *LGI provides no general mechanism for*

handling failed controllers; in particular, it provides no general mechanism for discovering and recovering failed controllers.

It should be pointed out that we are concerned mostly about Byzantine failures, i.e., failures that make controllers behave arbitrarily, unpredictably, and potentially maliciously. In fact, LGI can effectively tolerate fail-stop failures, i.e., failures that make controllers stop functioning and produce no illegal output: if a fail-stop failure occurs at a controller, when its interlocutors try to interact with it, they will get *exception* events, so that they can perceive and handle the failure according to their laws.

For some specific communities, it may be possible to eventually discover failed controllers, which generally requires a monitor agent who monitors and analyzes all interactions in the community, as shown in Law 2.2 (`monitor` law). However, this method relies on the honest self-reporting of the controllers on their interactive activities, while a failed controller may report different interactive activities than what it actually conducts. In addition, it may take a very long time for such an analysis to discover the failed controllers, and this latency may be exploited by the failed controllers to inflict massive damage. Furthermore, this method is not general: different communities may require different specifically tailored monitoring and analyzing methods.

There are several general methods for tolerating Byzantine failures, such as [8], [48], and [25]. However, these methods may incur very high replication cost and multicasting overhead for tolerating even one failed controller, while the number of controllers and the number of operations these controllers may execute in even a single community can be very large, let alone multiple communities. For most applications of LGI, these methods are far too inefficient and expensive; therefore, they cannot effectively remove LGI's Achilles' Heel.

Removing LGI's Achilles' Heel is one of the major goals of this work. In the following chapters, we will introduce our approach that can effectively remove LGI's Achilles' Heel.

Chapter 4

The Architecture of Cop

Cop is an LGI-based PEM for MAS. It inherits the low latency, high scalability, and regulated interoperability of LGI. In addition, it removes the Achilles' Heel of LGI, making it also highly dependable and secure.

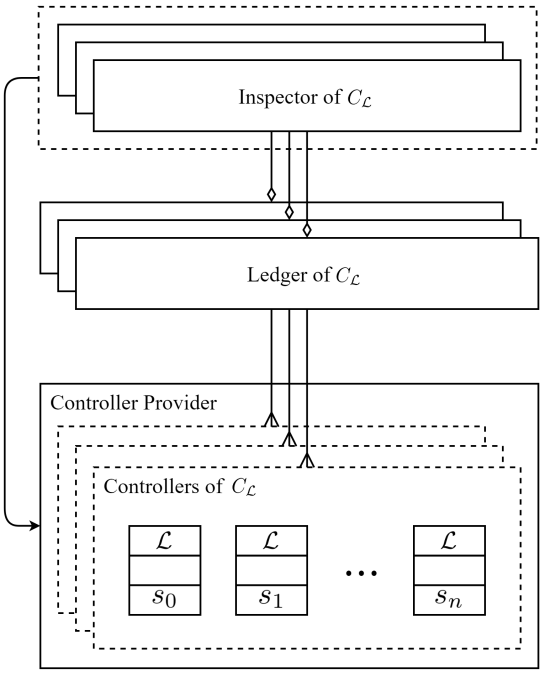
Cop consists of two parts working in concert:

- LGI. As an LGI-based PEM, Cop enforces protocols with LGI. Since LGI has been sufficiently introduced in Chapter 2, this chapter will not elaborate more on LGI.
- A mechanism called Discovery and Recovery (D&R), with two purposes:
  - Promptly discovers any controller who fails to comply with its law right after the failure occurs.
  - Immediately recovers the failed controllers right after discovering them and assists with the reparation of the damage they may have inflicted.

In order for the D&R mechanism to carry out these purposes, in Cop, each community has a *Ledger* that keeps records of the interactive activities of the controllers in the community, and an *Inspector* that systematically inspects the community's Ledger to discover failed controllers and initiates the recovery of the failed controllers. To cooperate with the Ledger and Inspector of each community, a component called *Controller Provider (CP)* takes the role of LGI's CoS; in addition to hosting controllers for multiple communities with enhanced security, CP is also responsible for feeding every community's Ledger and actuating the recovery of failed controllers. Figure 4.1 shows the overview of the architecture of Cop.

Ledgers, Inspectors, and CP are the Trusted Computing Base (TCB) of Cop, the correctness of the D&R mechanism depends on their correct operation. As the following discussion

Figure 4.1: Overview of Cop



will show, trustworthiness and security are of high priority in their design. Moreover, their trustworthiness and security can be further enhanced via traditional methods such as TPM.

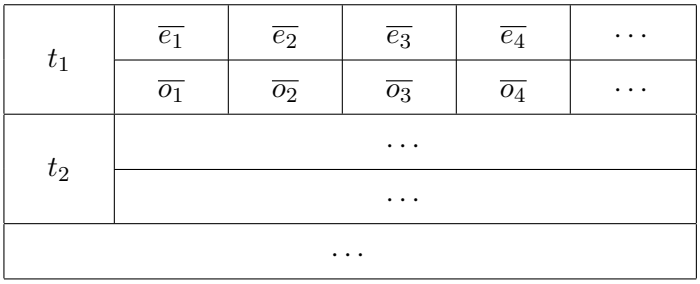
The rest of this chapter introduces the details of Ledger, Inspector, CP, and their relationship. With this chapter's introduction as a foundation, Chapter 5 will introduce how the D&R mechanism works.

4.1 Ledger

In Cop, each community has a structure called Ledger which keeps records of the interactive activities of each controller in the community, including events that occurred at each controller, and operations executed by each controller. The Ledger of the community $C_{\mathcal{L}}$ is denoted by $G_{\mathcal{L}}$.

As mentioned in Section 2.1, every event/operation has a unique byte string representation. We call the representation of an event an *event record*, and the representation of an operation an *operation record*. The event record representing the event e is denoted as $\overline{e}$, and the operation record representing the operation o is denoted as $\overline{o}$. A Ledger consists

Figure 4.2: Schematic of a Ledger



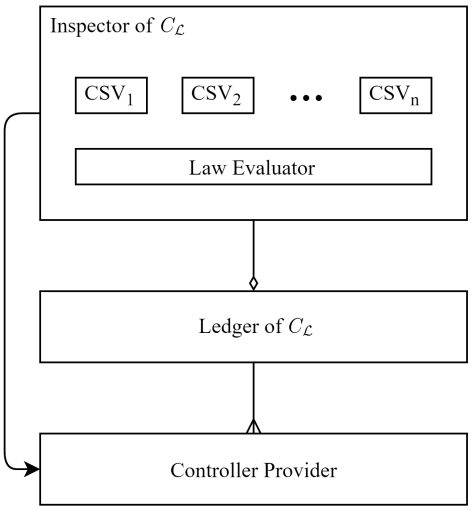
of event records and operation records, each of which is called a *Ledger Entry*. Ledger $G_{\mathcal{L}}$ maintains two sequences for each controller t in $C_{\mathcal{L}}$: $\overline{E_t}$, the sequence of event records for the events that occurred at t in their occurrence order; and $\overline{O_t}$, the sequence of operation records for the operations that t executed in their execution order. Figure 4.2 shows the schematic of a Ledger.

For the reason that will be discussed in Section 5.1.1, the Ledger does not keep records of *exception* events or *set*/*unset* operations. The events/operations recorded on a Ledger are all caused by interactive activities, or more precisely, network I/O. Some operation records represent failed operations and will be used in the D&R mechanism for inferring *exception* events. The events/operations for Enforced Obligations require some additional discussion, which will be provided in Section 7.1.

After the on-line protocol enforcement of $\mathcal{L}$ by any controller t in $C_{\mathcal{L}}$, the interactive activities of t will be recorded on $G_{\mathcal{L}}$, which will be used off-line as the basis of the D&R mechanism. Obviously, the accuracy and security of the Ledger of each community are crucial to the correctness of the D&R mechanism. Since the controllers may fail, the production of Ledger Entries cannot rely on the controllers' self-reporting. Chapter 6 will introduce our solution for reliably producing Ledger Entries based on the analysis of network I/O. In addition, for better security, the Ledger of each community is highly secluded, and is only accessible by the Inspector of the community and the Ledger Writers which will be introduced in Section 4.3.1.

Ledger is highly shardable. Since LGI controllers work locally and independently, Ledger entries of different controllers can be stored in different locations. Sharding a Ledger may reduce the risk of single point of failure or attack and provide better scalability.

Figure 4.3: Inspector of $C_{\mathcal{L}}$



Another seemingly possible way for organizing the Ledger is to use a single sequence to record the interactive activities of each controller t, so that whenever t processes an event, the corresponding event record will be appended to the sequence, and whenever t executes an operation, the corresponding operation record will be appended to the sequence. However, it is impractical to construct a Ledger organized in this way, as it depends on the time when the controller processes the event: as mentioned in Section 2.2, due to the atomic nature of the execution of operations, the time when an event occurs at a controller does not always equal to the time when the controller processes it; although as Chapter 6 will show, the time when an event occurs at a controller can be accurately figured out without needing to trust the controller, the time when the controller processes the event cannot be figured out without relying on the controller's self-reporting which is untrustworthy, as the processing of an event is an internal activity of the controller.

4.2 Inspector

For each community $C_{\mathcal{L}}$, Cop maintains an Inspector $I_{\mathcal{L}}$ that systematically and continuously inspects the community's Ledger $G_{\mathcal{L}}$ to discover failed controllers and initiate the recovery of failed controllers. As shown in Figure 4.3, the way $I_{\mathcal{L}}$ works, which will be elaborated in Chapter 5, is based on two components it maintains:

- Controller State Variable (CSV) for every controller t of $C_{\mathcal{L}}$, denoted as v_t, which is a variable reflecting the correct control state of t. Note that v_t is physically independent from the control state stored internally in t, and must be computed by $I_{\mathcal{L}}$: for each controller t, v_t is initially empty, and will be updated recursively during D&R, as Section 5.1.1 will show.

- Law Evaluator. It takes one event record $\overline{e}$ and one control state s as input, then evaluates $\mathcal{L} : E \times S \to O^* \times S$ with the corresponding event e and s and yields a sequence of operations O and the new control state s' that $\mathcal{L}$ mandates, and finally outputs the corresponding sequence of operation records $\overline{O}$ and s'.

Note that the Law Evaluator does not handle any network I/O, its only task is to evaluate the law; in particular, it does not execute the operations the law mandates. Therefore, the complexity of the Law Evaluator and the risk it faces are significantly lower than those of the controllers.

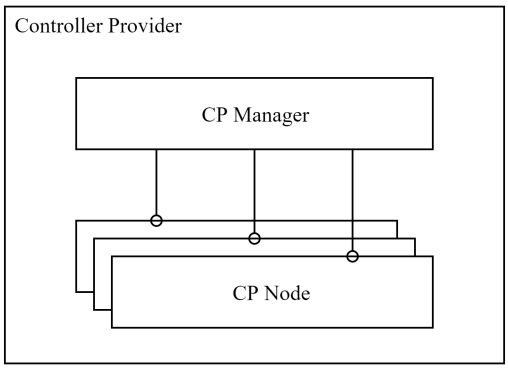
The local nature of controllers enables the CSVs of different controllers to be independent of each other. Meanwhile, the Law Evaluator does not record runtime state or rely on previous runtime state; therefore, it can be easily replicated and work in parallel. As Section 5.1.1 will show, the independent nature of CSVs and the high replicability of the Law Evaluator will bring considerable benefit to the scalability of the D&R mechanism.

The security of the Inspector of each community is crucial to the correctness of the D&R mechanism; therefore, each Inspector $I_{\mathcal{L}}$ is highly secluded and only has access to the components that are necessary to the D&R mechanism, i.e., the law $\mathcal{L}$, the Ledger $G_{\mathcal{L}}$, and CP with which $I_{\mathcal{L}}$ initiates the recovery of failed controllers.

4.3 Controller Provider

Controller Provider (CP) is a variant of LGI's CoS. In addition to hosting controllers for multiple communities, it also feeds every community's Ledger and actuates the recovery of failed controllers. It consists of a cluster of servers called *CP Nodes* and a component called *CP Manager* for managing the cluster. Figure 4.4 shows the overview of the architecture of CP.

Figure 4.4: Overview of Controller Provider



4.3.1 CP Node

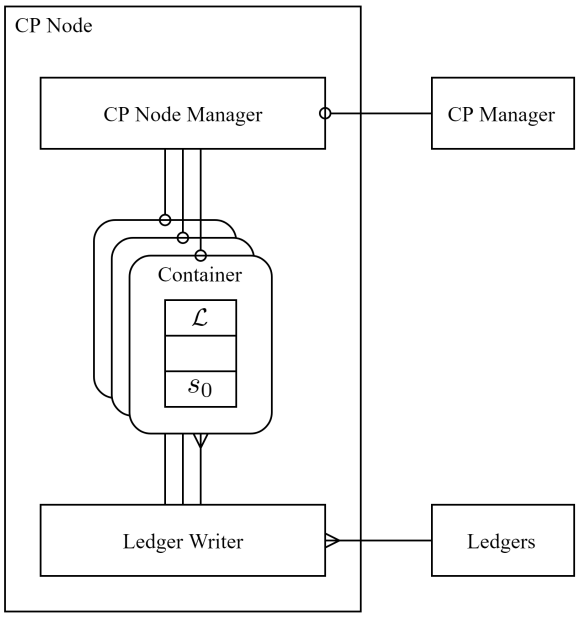
A CP Node is a server that hosts multiple generic controllers. The controllers on a CP Node may be loaded with different laws and thus may serve different communities. Figure 4.5 shows the architecture of a CP Node.

In a CP Node, controllers are encapsulated with Linux Containers (LXC) [30] [35], an OS-level virtualization mechanism for running multiple isolated instances of Linux system on a single host with a single Linux kernel. Two of the most important features of LXC include:

- Isolation. Powered by Linux namespaces [32], LXC can partition the resources of the host, including PIDs, UIDs, network, and file systems, and assign them to different containers, so that each container has an isolated and independent view of the environment that they can and can only operate in. On the other hand, with seccomp [32], LXC can also limit the system calls that the processes in a container can make. In this way, LXC can effectively prevent processes in a container from interfering with its cotenants on the same host, i.e., they cannot interfere with the processes in other containers or processes not in container.

- Resource control. Powered by Linux control groups (cgroups) [32], LXC can limit, prioritize, account, and control the resource usage of each container, including CPU, memory, storage, and network. In this way, LXC can effectively prevent processes in

Figure 4.5: CP Node



one container from hogging all the resources of the host; it can also help optimize the resource allocation to the containers dynamically to fit the actual demand.

By isolating controllers in containers, a CP Node can ensure that failed controllers cannot hog all the hardware resources or interfere with the environment outside the container, and thus prevent them from disabling or corrupting other controllers or components on the same CP Node. It also enables the CP Node to dynamically adjust resource allocation to the controllers it hosts to fit the actual demand.

It is noteworthy that Virtual Machine (VM) is another potential solution for isolating controllers. Theoretically, VM can provide more secure isolation than LXC, as the processes isolated in different VMs do not share the same kernel. However, as many works including [18] [24] [42] have pointed out, VM is generally less efficient and scalable than LXC. In addition, it is inflexible to dynamically adjust resource allocation with VM. Moreover, no work has convincingly demonstrated that LXC is really less secure than VM. Therefore, in this work, we chose LXC over VM.

Every CP Node has a component called *CP Node Manager* that manages the CP Node

and the controllers it hosts. It is also responsible for actuating the recovery of failed controllers, which will be elaborated in Section 5.2. For better security, the CP Node Managers only take input from the CP Manager. More on the CP Node Manager and the CP Manager will be discussed in Section 4.3.2.

In addition to hosting controllers, the CP Nodes are also responsible for feeding the Ledgers. The production of Ledger Entries cannot rely on the self-reporting of controllers, because controllers may fail and thus cannot be trusted to honestly report their interactive activities. Fortunately, each CP Node can capture the network I/O of the controllers it hosts at a lower level, enabling our solution for reliable Ledger Entry production that will be elaborated in Chapter 6, which is carried out by a component of the CP Node called *Ledger Writer*. It is noteworthy that the multiple communities that Cop may serve have a many-to-many relationship with the CP Nodes: a CP Node may host controllers for multiple communities, while a community may scatter over multiple CP Nodes. Therefore, a CP Node may need to feed the Ledgers for multiple communities, and the Ledger of one community may be fed by multiple CP Nodes.

The security of the Ledger Writer and the CP Node Manager of every CP Node is crucial to Cop. In fact, protecting the Ledger Writer and the CP Node Manager from the interference of failed controllers is another reason for isolating controllers in containers.

4.3.2 CP Manager

CP Manager is a CP component for managing the CP Nodes; it is also the interface for the users to use Cop. It has three main tasks:

- Routine management of controllers, including creating, removing, adjusting, and monitoring controllers.
- Monitoring the status of CP Nodes, including their health, load, and available hardware resource.
- Actuating the recovery of failed controllers when informed by an Inspector. More on this will be discussed in Section 5.2.

As Figure 4.5 shows, the management of the CP Manager is carried out via the CP Node Manager of every CP Node. Whenever the CP Manager needs to carry out a management operation, it will send a command message to the relevant CP Node Manager. For better security, the command messages will be sent via TLS. Upon receiving a command message, the CP Node Manager first checks the sender's certificate; then if and only if the certificate shows that the sender is the CP Manager, the CP Node Manager will execute the operation on the CP Node according to the command message and respond if necessary.

With a global view of all CP Nodes and controllers, the CP Manager can help optimize the utilization of the CP Nodes and the distribution of controllers among the CP Nodes. For example, it can dynamically adjust the resource allocation to the controllers on a CP Node to better fit the demand; it can also create new controllers on less occupied CP Nodes.

It is noteworthy that the CP Manager does not keep any runtime state, all the runtime state (e.g., the roster of controllers) is kept by the CP Node Managers. Therefore the CP Manager can be easily replicated to accommodate large numbers of CP Nodes and users.

Chapter 5

The D&R Mechanism

Cop ensures its dependability and security by removing the Achilles' Heel of LGI with the D&R mechanism, which carries out the following two complementary tasks based on the systematical and continuous inspection of the interactive activities of each controller in each community:

- Promptly discovers any controller who fails to comply with its law right after the failure occurs.
- Immediately recovers the failed controllers right after discovering them and assists with the reparation of the damage they may have inflicted.

This chapter introduces how the D&R mechanism works. It first introduces the mechanism for discovering failed controllers, then it introduces the mechanism for recovering failed controllers.

5.1 The Discovery Mechanism

Thanks to the local nature of LGI laws and their enforcement, the failure of a controller t in a community $C_{\mathcal{L}}$ to comply with $\mathcal{L}$ can be discovered locally by inspecting the interactive activities of t alone and independently from any other controllers. This significantly simplifies the inspection and makes it highly efficient and scalable. This section discusses the Discovery mechanism. For the sake of simplicity, our discussion will first focus on a single isolated community $C_{\mathcal{L}}$, whose members only interact with members of $C_{\mathcal{L}}$. Obviously, the Discovery mechanism can be carried out in parallel in multiple isolated communities in the same way, enabling it to support multiple isolated communities. Furthermore, our discussion will

be expanded to explain how the Discovery mechanism handles multiple communities that interoperate with each other.

5.1.1 For Isolated Communities

In a community $C_{\mathcal{L}}$, the inspection of a controller by the Inspector $I_{\mathcal{L}}$ starts when its first event record appears on the Ledger. This event must be an *adopted* event which occurs when the controller gets loaded with $\mathcal{L}$ and adopted by an actor, marking the start of its lifetime. When $I_{\mathcal{L}}$ observes an *adopted* event record $\overline{e_1}$ of a controller t on the Ledger $G_{\mathcal{L}}$, it carries out the *initial inspection* of t as follows:

1. $I_{\mathcal{L}}$ creates the CSV for t, denoted as v_t. Initially v_t is an empty set, the same as any newly spawned controller.
2. $I_{\mathcal{L}}$ evaluates $\mathcal{L}$ with the Law Evaluator for the pair $(\overline{e_1}, v_t)$, and gets the sequence of operation records $\overline{O_1}$ (except *set*/*unset* operations), and a control state s_1.
3. $I_{\mathcal{L}}$ sets v_t to s_1.
4. $I_{\mathcal{L}}$ then inspects $\overline{O_t}$ (the operation records of t on $G_{\mathcal{L}}$, see Section 4.1) to verify whether t has executed the exact sequence of operations mandated by $\mathcal{L}$ sequentially and atomically, i.e., $\forall i \in \left[1..\left|\overline{O_1}\right|\right], \overline{O_t}(i) = \overline{O_1}(i)$. Then:
   - If t has executed the exact sequence of operations mandated by $\mathcal{L}$ sequentially and atomically, then $I_{\mathcal{L}}$ can conclude that t has operated correctly so far.
   - Otherwise, $I_{\mathcal{L}}$ concludes that t is failed, and will initiate the recovery of t as Section 5.2 will describe.

After the initial inspection, $I_{\mathcal{L}}$ continuously carries out the *recursive inspection* of the consecutive event records in $\overline{E_t}$ (the event records of t on $G_{\mathcal{L}}$, see Section 4.1) until it observes a *quit* operation record:

1. Suppose $I_{\mathcal{L}}$ has inspected a sequence of event records $(\overline{e_1}, \overline{e_2}, \ldots, \overline{e_k})$ without detecting any failure, and for the events represented by these event records, $\mathcal{L}$ totally mandates

p operations whose corresponding operation records have all been inspected during the inspection. Then there are the following three situations:

- There is no more uninspected event record or uninspected operation record on $G_{\mathcal{L}}$. This situation means t does not conduct any more interactive activity; therefore, nothing should be done until more event records or operation records appear on $G_{\mathcal{L}}$.
- There is no more uninspected event record on $G_{\mathcal{L}}$, but there are some uninspected operation records on $G_{\mathcal{L}}$. This situation means the controller has executed more operations than $\mathcal{L}$ mandates; therefore, it should be regarded as failed, and $I_{\mathcal{L}}$ will initiate the recovery of t as Section 5.2 will describe.
- There are uninspected event records on $G_{\mathcal{L}}$. In this situation, $I_{\mathcal{L}}$ will go to Step 2 and start inspecting the next event.

2. Suppose the event record after $\overline{e_k}$ is $\overline{e_{k+1}}$. $I_{\mathcal{L}}$ evaluates $\mathcal{L}$ with the Law Evaluator for the pair $(\overline{e_{k+1}}, v_t)$, and gets the sequence of operation records $\overline{O_{k+1}}$ (except *set*/*unset* operations), and a control state s_{k+1}.
3. $I_{\mathcal{L}}$ sets the value of v_t to s_{k+1}.
4. $I_{\mathcal{L}}$ then inspects $\overline{O_t}$ to verify whether t has executed the exact sequence of operations mandated by $\mathcal{L}$ sequentially and atomically, i.e., $\forall i \in \left[1..\left|\overline{O_{k+1}}\right|\right], \overline{O_t}(p+i) = \overline{O_{k+1}}(i)$. Then:
   - If t has executed the exact sequence of operations mandated by $\mathcal{L}$ sequentially and atomically, then $I_{\mathcal{L}}$ can conclude that t has operated correctly so far.
   - Otherwise, $I_{\mathcal{L}}$ concludes that t is failed, and will initiate the recovery of t as Section 5.2 will describe.

The underlying philosophy of the inspection algorithm is: for a controller t in $C_{\mathcal{L}}$, no matter how it behaves internally, as long as it always executes the exact sequence of operations mandated by $\mathcal{L}$ sequentially and atomically for each event occurs at it, the effect of the controller on others is identical to the effect mandated by $\mathcal{L}$, and thus should be seen

as correct; otherwise, it should be seen as failed. Therefore, even though $I_{\mathcal{L}}$ does not and cannot access the internals of t, particularly the actual control state stored internally in t, it can still effectively verify the correctness of t. This requires that the CSV of t always accurately reflects the correct control state of t as mandated by $\mathcal{L}$, which can be proved as follows:

Proof. This can be proved with induction.

Basis step: Initially the CSV of t, denoted as v_t, is an empty set, which accurately reflects the correct control state of the newly spawned t. During the initial inspection, v_t is set to the control state mandated by $\mathcal{L}$; therefore, it still accurately reflects the correct control state of t.

Inductive step: Suppose that after a sequence of events $(e_1, e_2, \ldots, e_k)$, the correct control state of t is s_k, and v_t reflects the correct control state of t, i.e., $v_t = s_k$. When a new event e_{k+1} occurs at t, if t is correct, it should evaluate $\mathcal{L}$ which maps the pair (e_{k+1}, s_k) to (O, s_{k+1}), then execute the sequence of operations O and update its control state to s_{k+1}. On the other hand, according to the inspection algorithm, when $I_{\mathcal{L}}$ sees the event record $\overline{e_{k+1}}$, it evaluates $\mathcal{L}$ for the pair $(\overline{e_{k+1}}, v_t)$, which equals to $(\overline{e_{k+1}}, s_k)$, and gets the pair $(\overline{O}, s_{k+1})$, then updates v_t to s_{k+1}. Now $v_t = s_{k+1}$, i.e., it still reflects the correct control state of t.

In conclusion, v_t always accurately reflects the correct control state of t. □

Note that during the inspection, *set*/*unset* operations are ignored, as they will be reflected in the new control state. If a controller fails to execute the correct *set*/*unset* operations, it will have a wrong control state which will eventually lead to wrong operations, so that the Inspector can eventually discover this failure.

As mentioned in Section 4.1, some operation records on $G_{\mathcal{L}}$ represent failed operations. In a correct controller, a failed operation should incur an *exception* event with the information of the failed operation. Therefore, when observing a failed operation, $I_{\mathcal{L}}$ will imitate a correct controller and generate an *exception* event record accordingly. Before inspecting the next event record, $I_{\mathcal{L}}$ will first inspect the *exception* event records it has generated for the failed operations records during the inspection of the current event record in the

same order as the failed operations records. The reason behind this will be elaborated in Section 7.2.2. Sometimes it is impossible for $I_{\mathcal{L}}$ to get the information of a failed operation from $G_{\mathcal{L}}$, e.g., when the operation failed during the TCP handshake and thus no message has been exchanged at all. In this situation, $I_{\mathcal{L}}$ can generate the *exception* event with the operation record $\overline{o}$ which it computes for comparing with the failed operation, because if the controller is correct, the operation it failed to execute must be o.

The inspection algorithm is highly efficient, as it is on a linear and rolling basis, i.e., there is no look-ahead or look-behind: during the inspection of an event record, there is no need to consider any future event records, and same to operation records; on the other hand, the inspected Ledger Entries will not be inspected again. The high efficiency of the inspection algorithm enables $I_{\mathcal{L}}$ to discover the failed controllers promptly after the Ledger Entries reflecting the incorrect activity appear on $G_{\mathcal{L}}$.

In addition, since each controller works locally, their Ledger Entries can be inspected independently in parallel, making the inspection algorithm highly parallelizable. Moreover, as mentioned in Section 4.2, the independent nature of the CSVs and the high replicability of the Law Evaluator make it possible for a community to have multiple Inspector instances working in parallel, so that each instance only carries out the Discovery mechanism for a subset of controllers of the community, which makes the inspection algorithm highly scalable.

The same inspection algorithm can be executed in parallel in multiple isolated communities, enabling the Discovery mechanism to support multiple isolated communities.

5.1.2 For Interoperating Communities

The inspection algorithm introduced in Section 5.1.1 can also support interoperating communities. This is because, as mentioned in Section 2.3, in LGI, interoperations between two communities are locally governed by the laws of these communities; therefore, any controller t' in a community that interoperates with another community still needs to operate subject to the law of its community, in which sense it is not different from any controller t in an isolated community; therefore, the inspection algorithm for t' is exactly the same as the inspection algorithm for t.

5.2 The Recovery Mechanism

When a failed controller is discovered, it will be immediately recovered so that its operation can be put back to normal. Since controllers are independent of each other, the recovery of an individual failed controller can be done without considering other controllers, making it highly efficient and scalable.

The Recovery mechanism has the following two tasks:

- Resumes the proper operation of the failed controller.
- Assists with the reparation of the damage that the failed controller may have inflicted.

5.2.1 Resuming the Proper Operation

Resuming the proper operation of a failed controller may seem to be a daunting task, as it seems to suggest the need to examine the entire runtime context of the failed controller to find what is wrong and try to fix the problem. This procedure is also known as "debugging", which is notoriously labor-intensive and time-consuming, as different failures may have different causes and require different and specific examinations and solutions. Fortunately, thanks to the generic nature of controllers, such a debugging is unnecessary: the code of controllers is identical, what makes them different is their laws and control states. Therefore, to resume their proper operation, we can just respawn the controller with the authentic controller code, then load it with the law and the correct control state which can be retrieved from its CSV.

The resumption of the proper operation of a failed controller t in a community $C_{\mathcal{L}}$ is efficiently done as follows:

1. When the Inspector $I_{\mathcal{L}}$ discovers a failed controller t, it will send a recovery notification to the CP Manager with t's location and its CSV value v_t via TLS.
2. Upon receiving a recovery notification, the CP Manager will first check the sender's certificate; if and only if the certificate shows that the sender is an Inspector, the CP Manager will locate t and send a command message to the CP Node Manager of the CP Node that hosts t, denoted as N_t.

3. N_t respawns t with the authentic generic controller code.

4. N_t loads the respawned t with the law $\mathcal{L}$.

5. N_t loads v_t to t. Now the proper operation of t is resumed.

Since controllers are independent of each other, the resumption of failed controllers can be done in parallel, making the Recovery mechanism highly scalable.

5.2.2 Repairing the Damage

Although failed controllers can be promptly discovered after they behave illegally, there is still some latency in which the failed controllers may inflict damage on others. Therefore, after the resumption of the operation of a failed controller, it is also necessary to repair the damage it may have inflicted.

As mentioned in Chapter 3, a failed controller may violate its law in the following ways:

- *Minus Case*: it may have not executed some operations that its law mandates.
- *Plus Case*: it may have executed some operations that its law does not mandate.
- It may have executed a different sequence of operations than what its law mandates. This is essentially a combination of the Minus Case and the Plus Case.

After discovering a failed controller, according to the Ledger, the Inspector will generate a *diagnosis report* for the controller indicating the incorrect operations it has executed for the event which marks its failure, the operations that it has not executed but should have for that event, and the events occurred at it and the operations it has executed after the failure happened. With this diagnosis report, the Minus Case can be largely handled: after resuming the proper operation of a failed controller t, N_t will direct it to execute the unexecuted operations for the event it wrongly processed, then re-process the events that occurred after its previous failure, without repeating the already executed operations.

For the Plus Case, the diagnosis report of the failed controller can also assist with the reparation of the damage, as it provides us with a degree of understanding of the nature of the damage so that we can attempt to fix it accordingly. However, sometimes the operations

a failed controller has executed may have cascading or irreversible effects. For example, consider Law 2.1 (`money` law), suppose a failed controller t sent m amount of money to another controller t' although it only had the budget of 0, even if t is promptly discovered and recovered, the m amount of money has already been illegally sent to t', and t' may, in turn, send this illegal money to others legally. In these situations, it may be necessary to examine the Ledger Entries of all involved controllers to get a comprehensive understanding of the nature of the damage in order to attempt to repair the damage. The systematical handling of the Plus Case is beyond the scope of this work.

Chapter 6

Ledger Entry Production

For the D&R mechanism to work correctly, it is crucial that the Ledger accurately reflects the interactive activities of the controllers in its community. This requires a reliable mechanism for producing Ledger Entries.

The seemingly most straightforward way is to require every controller to report all its events/operations to the Ledger. However, this method is unreliable, as the controllers may fail and cannot be trusted to honestly report their events/operations.

Fortunately, in a proper LGI implementation, almost all event/operation types have distinct network I/O patterns, and thus can be accurately inferred from the network I/O. There are only three event/operation types that cannot be directly inferred from the network I/O:

- *set*/*unset* operations. As discussed in Section 5.1.1, they do not need to appear on the Ledger, and thus do not need to be considered.
- Events/operations about Enforced Obligations, i.e., *imposeObligation*, *repealObligation*, and *obligationDue*. Section 7.1 will introduce our method for making them inferable.
- *exception* events. As discussed in Section 4.1, *exception* events are caused by failed operations and are recorded as failed operation records on the Ledger. In LGI, all operations that may cause *exception* events are inferable, making *exception* events also inferable. More details will be discussed in Section 6.2.

No matter how a controller behaves internally, its network I/O always objectively reflects its interactive activities, which is all that the Ledger concerns. With all relevant

events/operations directly or indirectly inferable from the network I/O, Ledger Entries can be reliably produced by the inference. Note that the inference has some requirements on the implementation of LGI, which will be elaborated in Chapter 7.

6.1 Event Record Production

Except for *exception* events which will be discussed in Section 6.2, every event that occurs at a controller is caused by a *network input stream* to the controller, i.e., a TCP stream initiated by a peer of the controller. An event can be inferred from a network input stream, in particular, the TCP handshake information, the peer's information, and the messages exchanged in the stream, then an event record representing this event can be naturally produced.

Different LGI implementations may lead to different network input stream patterns for the same event type. Appendix B lists the patterns of all events in our reference LGI implementation.

Sometimes it may be impossible to infer an event from a network input stream. There are two possible causes of this situation:

- The controller did not follow the interactive protocol of the LGI implementation. In this case, the controller should be directly reported as failed.
- The controller followed the interactive protocol of the LGI implementation, but something else went wrong, e.g., the stream was broken, timeout happened, or the peer did not follow the interactive protocol. In a proper LGI implementation, such a network input stream should not cause an event; therefore, during the inference, it is safe to just ignore it.

6.2 Operation Record Production

Except *set/unset* operations which do not need to appear on the Ledger, every operation executed by a controller will cause a *network output stream* from the controller, i.e., a TCP stream initiated by the controller. An operation can be inferred from a network output

stream, in particular, the TCP handshake information, the peer's information, and the messages exchanged in the stream, then an operation record representing this operation can be naturally produced.

Different LGI implementations may lead to different network output stream patterns for the same operation type. Appendix B lists the patterns of all operations in our reference LGI implementation.

Sometimes it may be impossible to infer an operation from a network output stream. There are two possible causes of this situation:

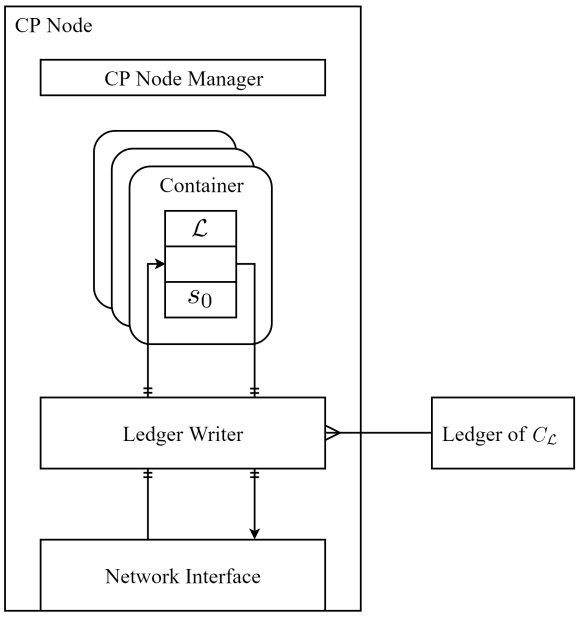
- The controller did not follow the interactive protocol of the LGI implementation. In this case, the controller should be directly reported as failed.
- The controller followed the interactive protocol of the LGI implementation, but something else went wrong, e.g., the stream was broken, timeout happened, or the peer did not follow the interactive protocol. In a proper LGI implementation, such a network output stream should cause an *exception* event; therefore, in this case, a failed operation record should be produced.

6.3 Ledger Writer

As Figure 6.1 shows, each CP Node has a component called Ledger Writer for producing Ledger Entries for all controllers the CP Node hosts. For each controller, the Ledger Writer captures its network I/O outside the container, and periodically stores the captured packets (maybe none) into batches; from each batch, the Ledger Writer infers events/operations, produces Ledger Entries, orders them by their TCP handshake completion timestamps (for event records, see Section 7.3) or TCP handshake start timestamps (for operation records, as they can reflect the time the controller executes the operations), then writes them to the Ledger of the controller's community.

Sometimes a batch may contain incomplete streams, i.e., streams without `FIN` or `RST` packets. Such streams are not necessarily broken streams: maybe they were just cut into pieces stored in multiple batches. In order to handle this situation, whenever the first incomplete stream is observed in a batch, the Ledger Writer will stop processing the rest

Figure 6.1: Ledger Writer



of the batch; instead, it will prepend the unprocessed packets in the current batch to the next batch and move on to processing the next batch.

The way the Ledger Writer handles incomplete streams has one seeming vulnerability. In a proper LGI implementation, hanging connections caused by abruptly irresponsive peers are handled with the timeout mechanism: for a network input stream which is supposed to cause an event, if it takes longer than the allowed time, the controller should terminate it and no event will occur; for a network output stream which is caused by an operation, if it takes longer than the allowed time, the controller should terminate it, and an *exception* event should occur. When a proper controller terminates a timeout stream, it will send an `RST` packet which marks the completion of the stream, so that the Ledger Writer can move on to processing other streams. However, controllers may fail and may never send the `RST` packet, making the Ledger Writer believe this is an incomplete stream and waits forever for it to complete. In this situation, the Ledger Writer is practically disabled.

To solve this problem, in each batch, the Ledger Writer records its creation timestamp. For each stream, if the difference between the batch creation timestamp and the stream's TCP handshake completion timestamp is more than the allowed time, i.e., at the time the

batch was created the stream had already used up its allowed time, then the stream must be a timeout stream. If the stream is a network input stream, then the Ledger Writer will just ignore it; if it is a network output stream, then the Ledger Writer will create a failed operation record. After processing the timeout stream in such a way, the Ledger Writer will move on to processing other streams.

It is noteworthy that most network streams the Ledger Writer handles are TLS streams. In order to infer events/operations, the Ledger Writer needs to have the session keys of the streams. In TLS, session keys are negotiated by both sides with secure key exchange algorithms such as Diffie-Hellman [14] or RSA [50]; therefore, it is impossible to retrieve the session keys from the TLS streams themselves. In order to meet the requirement under this constraint, every controller is required to export the session keys and encryption algorithm names of all TLS streams it handles right after the TLS handshake to a location on the CP Node only accessible by the Ledger Writer, so that the Ledger Writer can decrypt the payload. After decrypting the payload, the Ledger Writer will remove the session keys and the encryption algorithm names. If the Ledger Writer cannot decrypt the payload of a complete TLS stream of a controller with the session key the controller provides, it means the controller did not correctly or honestly export the session key or the encryption algorithm name, then the controller should be directly reported as failed.

The way the Ledger Writer retrieves session keys has one seeming vulnerability: a failed controller may have actually sent message m with key k, which yielded ciphertext c in the TLS stream, but it wants to mislead the Ledger Writer into believing that it has sent message m', so it computes and reports a key k', with which m' can be encrypted as c as well; then when the Ledger Writer decrypts c with k', it will get m', and thus be misled. However, this situation is virtually impossible, because it is essentially a Known-Plaintext Attack (KPA) with only one plaintext-ciphertext pair, while as pointed out by [57], TLS encryption algorithms are required to be strong against KPA.

Since the production of Ledger Entries for each controller is independent of others, for better performance, the Ledger Writer can be replicated and work in parallel, so that each Ledger Writer instance only needs to produce Ledger Entries for a subset of controllers.

Chapter 7

Cop's Requirements on LGI's Implementation

To ensure the correctness of the D&R mechanism, Cop has several requirements on LGI's implementation. To meet these requirements, we made several enhancements to the original LGI implementation called Moses. The enhanced LGI implementation is called *Moses-2*. This chapter discusses these requirements and how Moses-2 addresses them.

7.1 Observable Obligations

As introduced in Section 2.1, Enforced Obligation is one of the essential features of LGI. In the original LGI implementation, obligations of a controller are maintained by the controller internally in the following way:

- When an *imposeObligation* operation is executed, the controller will create an internal timer for the obligation.
- When the time is up, an *obligationDue* event will occur internally at the controller.
- When a *repealObligation* operation is executed, the controller will remove the internal timer of the specified obligation.

However, in this way, *imposeObligation*, *repealObligation*, and *obligationDue* will not cause any network I/O, making the Ledger Writer unable to infer and record them on the Ledger. Therefore, Cop requires *a mechanism that makes obligations observable.*

To meet this requirement, Moses-2 handles obligations with a component called *Obligation Registry* running on each CP Node, which manages obligations for all Moses-2 controllers on the CP Node in the following way:

- When a Moses-2 controller executes an *imposeObligation* operation, it will send an *imposeObligationRequest* message with the obligation's name, arguments, and delay to the Obligation Registry of its CP Node. Upon receiving a valid *imposeObligationRequest* message from a Moses-2 controller on its CP Node, the Obligation Registry will create a timer of the specified delay for the obligation, then respond with an *imposeObligationResponse* message without errors indicating the successful imposition of the obligation. Failing to receive such a response will cause an *exception* event on the Moses-2 controller.

- When the time is up, the Obligation Registry will send an *obligationDueMessage* message with the obligation's name and arguments to the Moses-2 controller that imposed the obligation. Upon receiving a valid *obligationDueMessage* message from the Obligation Registry of its CP Node, a corresponding *obligationDue* event will occur at the Moses-2 controller.

- When a Moses-2 controller executes a *repealObligation* operation, it will send a *repealObligationRequest* message with the obligation's name to the Obligation Registry of its CP Node. Upon receiving a valid *repealObligationRequest* message from a Moses-2 controller on its CP Node, the Obligation Registry will remove the timer for the obligation, then respond with a *repealObligationResponse* message without errors indicating the successful repeal of the obligation. Failing to receive such a response will cause an *exception* event on the Moses-2 controller.

In this way, all obligation-related events/operations will cause network I/O, enabling Ledger Writer to infer and record them on the Ledger.

For better security, all interactions between Moses-2 controllers and the Obligation Registry are through TLS. Before a Moses-2 controller sends an *imposeObligationRequest* or *repealObligationRequest* message, it will check the receiver's certificate and only send the message if the certificate shows that the receiver is the Obligation Registry. On the other hand, when a Moses-2 controller receives an *obligationDueMessage* message, it will check the sender's certificate and only trigger an *obligationDue* event if the certificate shows that the sender is the Obligation Registry.

7.2 Consistent Event Processing Order

In order for the Ledger to correctly reflect the interactive activities of a controller, Cop requires:

1. *A mechanism that ensures that the order of the events that the controller actually processes is consistent with the order of the events recorded on the Ledger.*

2. *A mechanism that ensures that the order of the operations that the controller actually executes is consistent with the order of the operations recorded on the Ledger.*

Requirement 2 is trivial because LGI requires operations to be executed sequentially and atomically. However, although seemingly trivial as well, Requirement 1 is actually quite tricky. The rest of this section will elaborate on the challenges of Requirement 1, and our solutions.

7.2.1 For Non-*exception* Events

As mentioned in Section 6.1, in LGI, except for *exception* events, all events that occur at a controller are caused by network input. The events recorded on the Ledger are ordered by their timestamps, which in Moses-2, due to the reason that will be discussed in Section 7.3.1, are their corresponding network input streams' TCP handshake completion timestamps. When a controller finishes reading a network input stream's payload, it will try to derive an event from the payload, and if a valid event is derived, the controller will append the event to an internal First-In-First-Out (FIFO) structure called *Event Queue*. Meanwhile, a processor thread in the controller keeps processing the events in the Event Queue in FIFO order.

To satisfy Cop's requirement of consistent event processing order, the most straightforward approach is to use one single thread to process network input in each controller, so that the TCP handshake of a network input stream will not be completed until its prior network input stream has been completely processed. In this way network input streams are naturally processed sequentially in the order of their TCP handshake completion timestamps. However, this will incur significant overhead: a network input stream may need a

very long time to be processed due to a large payload or a slow peer, and during this time all following network input streams have to wait. On the other hand, a network input stream must wait until all prior network input streams have been completely processed. With a large network input traffic, the overhead may become unacceptable.

It is more practical to use multiple threads to process network input. We can use one thread s to listen to the specified port and handles TCP handshakes, once a TCP handshake completes, a socket c will be created, then s will create a worker thread w, hand c to w, then continue with the next TCP handshake. After being created, w will read all payload from c, derive an event e from the payload, acquire the Event Queue's lock l_e, append e to the Event Queue, then release l_e. In this way, network input streams can be processed in parallel without needing to wait for others to finish. However, Cop's requirement of consistent event processing order may not be satisfied in the following situation.

Suppose there are two network input streams α and β, which complete TCP handshake at time t_α and t_β respectively, and $t_\alpha < t_\beta$, i.e., α completes TCP handshake earlier than β; α is processed by the worker thread w_α, and is supposed to cause event e_α; β is processed by the worker thread w_β, and is supposed to cause event e_β; α and β require the time of p_α and p_β to be processed respectively, and $p_\alpha > p_\beta$ because the payload of α is larger than β, Now consider the possible situation $t_\alpha + p_\alpha > t_\beta + p_\beta$, in which w_α appends e_α to the Event Queue later than w_β appends e_β. So e_α will be processed later than e_β by the controller. However, since $t_\alpha < t_\beta$, e_α is prior to e_β on the Ledger. Therefore in this situation, Cop's requirement of consistent event processing order is not satisfied.

Moses-2 provides an effective solution to this problem. A Moses-2 controller processes network input with multiple threads; in addition, it controls the event processing order with an internal FIFO structure called *Reservation Queue* in the following way:

1. In a Moses-2 controller, one thread s listens to the specified port and handles TCP handshakes. Once a connection is established and a socket c is created, s will create a worker thread w with the unique ID i_w, acquire the Reservation Queue's lock l_r, append i_w to the Reservation Queue, release l_r, and hand c to w, then continue with the next TCP handshake.

2. After being created, w will read all payload of c, create an event e according to the payload, then perform a procedure called *reservation check* defined as follows:

   (a) First w acquires l_r and checks if i_w is at the head of the Reservation Queue:

   - If i_w is not at the head of the Reservation Queue, then w will release l_r and be put to sleep.
   - Otherwise, w will acquire the Event Queue's lock l_e, append e to the Event Queue, release l_e, acquire l_r, remove i_w from the head of the Reservation Queue, wake up all sleeping worker threads, then release l_r, and thus its running is completed.

   (b) Upon being awakened, the worker threads will perform reservation check recursively.

With the Reservation Queue, Moses-2 satisfies Cop's requirement of consistent event processing order without losing the performance benefits provided by multiple threads. Take the aforementioned situation with network input streams α and β as an example. Since $t_\alpha < t_\beta$, i_{w_α} will be prior to i_{w_β} in the Reservation Queue. Therefore even though $t_\alpha + p_\alpha > t_\beta + p_\beta$, after reading the payload of β and producing e_β, w_β will be put to sleep as i_{w_β} is not at the head of the Reservation Queue. When w_α finishes reading the payload of α and produces e_α, it will append e_α to the Event Queue because i_{w_α} is at the head of the Reservation Queue. w_α will then remove i_{w_α} from the head of the Reservation Queue and wake up all sleeping worker threads, including w_β. Upon being awakened, w_β will find i_{w_β} at the head of the Reservation Queue, so it will append e_β to the Event Queue and remove i_{w_β} from the head of the Reservation Queue. In the end, e_α will be processed earlier than e_β by the controller, and Cop's requirement of consistent event processing order is thus satisfied.

7.2.2 For *exception* Events

Reservation Queue is effective for events caused by network input. However, *exception* events are not caused by network input; therefore, they cannot benefit from the Reservation Queue and need special treatment.

Before processing the next event, a controller should process the *exception* events (if any) in the same order as the execution order of the operations that cause them. The reason behind this is:

- As a common engineering rule of thumb, exceptions should be handled as soon as possible. In LGI, operations are atomically executed; therefore, the earliest possible time to process the *exception* event caused by an operation is right after the execution of all the operations the law mandates for the current event.
- In LGI, operations are sequentially executed; therefore, it is natural that the *exception* events they caused are processed in the same order as their execution order.

Moses-2 addresses this requirement with an internal FIFO structure called *Exception Queue* in the following way:

1. In a Moses-2 controller, operations are executed by a single processor thread which keeps processing the events in the Event Queue in FIFO order by sequentially and atomically executing the operations mandated by the law for each event. Before processing each event, the processor thread will clear the Exception Queue.
2. During processing an event, if any operation fails to execute, the processor thread will generate an *exception* event with the information of the failed operation, then append it to the Exception Queue.
3. After processing an event, if the Exception Queue is not empty, the processor thread will prepend all elements in the Exception Queue to the Event Queue.

In this way, the *exception* events caused in the current event will be processed before the next event in the same order as the execution order of the failed operations that cause them. Meanwhile, Cop's requirement of consistent event processing order is satisfied, as the order of *exception* events recorded on the Ledger is essentially the order of the corresponding failed operation records.

7.3 Consistent Event Timestamp

As mentioned in Section 2.2, in LGI, every event has a timestamp which may affect the rulings of the law. In order for the Ledger to correctly reflect the interactive activities of a controller, Cop requires:

- *The timestamp of each event that the controller actually uses when evaluating the law is consistent with the timestamp recorded on the Ledger.*
- *The event timestamps are consistent with their processing order.*

7.3.1 For Non-*exception* Events

As mentioned in Section 6.1, except for *exception* events, all events are caused by network input. As most network programs do, Moses-2 handles network input with sockets, specifically blocking sockets. Therefore, the earliest time a Moses-2 controller can access a network input stream is when the corresponding socket is created. As pointed out by [52], a blocking socket is created when the TCP connection is established, i.e., when the TCP handshake completes; therefore, it is most natural for Moses-2 to use the TCP handshake completion timestamp as the timestamp of each event. Moreover, the mechanism introduced in Section 7.2.1 ensures that the event timestamps generated in this way are consistent with their processing order.

On the other hand, a network input stream's TCP handshake completion timestamp can be easily observed and recorded in the corresponding Ledger Entry by the Ledger Writer, as it is the timestamp of the `ACK` packet of the three-way TCP handshake.

Therefore, for a non-*exception* event, Moses-2 uses the TCP handshake completion timestamp of the corresponding network input stream as its timestamp, so does the Ledger Writer.

7.3.2 For *exception* Events

Unlike other events, *exception* events are not caused by network input; they are caused by failed operations, which obviously cannot be executed before the occurrence of the event

for which the law mandates them. Therefore, the timestamps of *exception* events must not be earlier than the timestamp of the event in which they are caused. Moreover, due to the reason discussed in Section 7.2.2, the *exception* events caused by the operations mandated by the law for the current event should be processed before the next event; therefore, the timestamp of *exception* events must be earlier that than the timestamp of the next event.

The seemingly most straightforward way for setting the timestamp of an *exception* event is to use the timestamp when the corresponding failed operation is executed. However, due to the situation discussed in Section 7.2.1, this cannot guarantee that the timestamp of the *exception* event is earlier than the timestamp of the next event.

Therefore, Moses-2 uses the timestamp of the event in which the *exception* event is caused as the timestamp of the *exception* event. This way the timestamp of the *exception* event is not earlier than the timestamp of the current event and is always earlier than the timestamp of the next event. Although this will make the *exception* events caused in the same event have the same timestamp, the tie on their processing order can be easily broken with the execution order of the operations that cause them.

7.4 Treatment of Non-deterministic Laws

As introduced in Section 2.1, non-deterministic law is one of the essential features of LGI. In order for the Inspector to correctly inspect a controller loaded with a non-deterministic law, Cop requires *a mechanism that enables the Inspector to know each random number the controller generates*. Since the controller may fail, such a mechanism should not rely on the controller's self-reporting.

Moses-2 addresses this requirement as follows:

1. Every Moses-2 controller has a unique ID. During the initialization of a Moses-2 controller, the MD5 [49] hash of the concatenation of its unique ID and law text is computed and saved in its control state as the seed, call it r.

2. During processing an event, when the Moses-2 controller is about to generate the first random number, it retrieves r from its control state, and returns a pseudorandom number r_1 generated by a Linear Congruential Generator (LCG) [29] with the

arguments introduced in [47]:

$$r_1 = r \times 1372383749 + 1289706101 \pmod{2^{32}}$$

3. All following random numbers during the processing of the current event are generated recursively:

$$r_{k+1} = r_k \times 1372383749 + 1289706101 \pmod{2^{32}}$$

4. After processing the event, i.e., after the atomic execution of the operations mandated by the law for the event, the controller saves the last pseudorandom number it generates to its control state as the new value of r, then continues from Step 2.

With the same arguments and seed, LCG will generate the same sequence of pseudorandom numbers. For each controller, the Inspector has the access to its controller ID and the law text, so it can generate the same seed which is actually used by the controller. During the inspection, whenever the law requires a random number to be generated, the Inspector will follow similar procedures as described above, the only difference is: rather than loading r from the control state or saving r to the control state, the Inspector will load r from the CSV of the controller, and save r to the CSV of the controller. In this way, the Inspector can get each pseudorandom number generated by the controller; therefore, Cop's requirement on random numbers is satisfied.

Chapter 8

A Prototype of Cop and its Evaluation

We have developed a functionally complete prototype of Cop, in which the Inspector and the Ledger Writer are written in Python 3; Linux containers are managed with Docker [36]; network I/O is captured with tcpdump [23], and preliminarily analyzed with tshark [10]; CP Manager and CP Node Manager are written in Bash.

We have conducted several experiments with the prototype. Since the low latency, high scalability, and regulated interoperability of LGI have been extensively studied and verified in many works including [41], [39], and [2], our experiments mainly focused on evaluating the correctness and performance of the D&R mechanism.

It should be pointed out that due to various limitations, our experiments were conducted with weak hardware; in particular, the hard drives we used are very slow, the CPUs we used have mediocre single-core frequency, and the number of machines we could access is very small. The hardware limitation makes us unable to conduct larger scale experiments. Moreover, although our prototype is functionally complete, it is not optimized in terms of performance; in particular, the Inspector and the Ledger Writer of our prototype are written in Python, an interpreted scripting language with significantly lower performance than compiled languages such as C and C++. An industry-level implementation of Cop should have sufficient performance optimization and be deployed on a much more powerful hardware platform; therefore, it is expected to have a much higher performance than what our experiment results will show. Nonetheless, the results of our experiments still reveal some important properties of Cop that are expected to stay in an industry-level implementation. We will first discuss our experiments and the results, then in Section 8.4 we will propose a rational estimation of the performance of an industry-level implementation of Cop.

The rest of this chapter discusses our experiments and the results. Section 8.1 discusses

our experiments for validating the correctness of the D&R mechanism; Section 8.2 discusses our experiments for evaluating the performance of the Discovery mechanism; Section 8.3 discusses our experiments for evaluating the performance of the Recovery mechanism; Section 8.4 discusses the experiment results.

8.1 Correctness Evaluation of the D&R Mechanism

This set of experiments are for validating the correctness of the D&R mechanism of our prototype, i.e., no false positive: a correct controller should not be reported as a failed controller; and no false negative: a failed controller should not be reported as a correct controller.

Our experiments were conducted in 3 different settings:

- 1 CP Node running as a VirtualBox VM with eight 3.0 GHz virtual CPU cores, 8GB RAM, 7200 RPM HDD, and Debian OS. The CP Node hosts 10 controllers.
- 50 CP Nodes running as KVM VMs, each of which has one 2.2 GHz virtual CPU core, 2GB RAM, 5400 RPM HDD, Debian OS, and hosts 1 controller. There are 50 controllers in total.
- The same 50 CP Nodes, each of which hosts 2 controllers. There are 100 controllers in total.

We conducted the following experiments in each of the settings:

- All controllers follow Law 8.1 (`trivial` law) in the beginning: all messages the actor wants to send will be directly forwarded, and all arrived messages will be directly delivered to the actor. After 30 seconds, one of the controllers starts to behave abnormally by following Law 8.2 (`trivial-rogue` law), although during the handshakes it will still claim it is following Law 8.1: all messages the actor wants to send will be appended with a string "-rogue" and then forwarded; and all arrived messages will be appended with a string "-rogue" and then delivered to the actor. Each controller sends a random string to a random controller every $1 \sim 60$ seconds.

- All controllers follow Law 2.1 (`money` law) in the beginning: each controller is provided with an initial budget of $1000, then it can transfer to another controller any amount of money only if the amount is not greater than its current budget, after which the budgets of the sender and the receiver will be updated appropriately. After 30 seconds, one of the controllers starts to behave abnormally by following Law 8.3 (`money-rogue` law), although during the handshakes it will still claim it is following Law 2.1: whenever it sends money to others, the amount it sends is always its budget plus 1, and thus it violates Law 2.1. Each controller sends a random positive integer to a random controller every $1 \sim 60$ seconds representing money transfer.

- All controllers follow Law 8.4 (`throttle` law) in the beginning: each controller can only send 1 message per 60 seconds. After 30 seconds, one of the controllers starts to behave abnormally by following Law 8.1 (`trivial` law) which does not have this restriction, although during the handshakes it will still claim it is following Law 8.4. Each controller sends a random string to a random controller every $1 \sim 60$ seconds.

- In the beginning, all controllers follow Law 8.6 (`dummy-hierarchy` Conformance Hierarchy), whose root law is Law 8.5 (`root` law): the operations mandated by the root law and the subordinate law will be executed in turn: in the root law's turn, the controller will send a message containing the string "root" before sending the message that the actor wants to send; in the subordinate law's turn, the controller will send a message containing the string "sub" before sending the message that the actor wants to send. After 30 seconds, one of the controllers starts to behave abnormally by following Law 8.1 (`trivial` law) and will not send the "root" or "sub" messages, although during the handshakes it will still claim it is following Law 8.6. Each controller sends a random string to a random controller every $1 \sim 60$ seconds.

In each experiment, the failed controller was correctly discovered, and no correct controller was reported as a failed controller. After being discovered and recovered, all previously failed controllers behaved correctly. These experiments verify the correctness of the D&R mechanism of our prototype.

Law 8.1: `trival` law

```
Name: trivial
LawScript: CoffeeScript

UPON "sent", ->
  DO "forward"
  return true

UPON "arrived", ->
  DO "deliver"
  return true
```

Law 8.2: `trivial-rogue` law

```
Name: trivial-rogue
LawScript: CoffeeScript

UPON "sent", ->
  DO "forward", message: "#{@message}-rogue"
  return true

UPON "arrived", ->
  DO "deliver", message: "#{@message}-rogue"
  return true
```

Law 8.3: `money-rogue` law

```
Name: money-rogue
LawScript: CoffeeScript

UPON "adopted", ->
  DO "set", key: "budget", value: 1000
  return true

UPON "sent", ->
  DO "forward", message: CS("budget") + 1
  return true

UPON "arrived", ->
  DO "set", key: "budget", value: CS("budget") + @message
  DO "deliver"
  return true
```

Law 8.4: `throttle` law

```
Name: throttle
LawScript: CoffeeScript

UPON "adopted", ->
  DO "set", key: "throttled", value: false
  return true

UPON "arrived", ->
  DO "deliver"
  return true

UPON "sent", ->
  if not CS("throttled")
    DO "set", key: "throttled"
    DO "impose_obligation", name: "unthrottle", delay: 60000
    DO "forward"
  return true

UPON "obligation_due", ->
  if @name is "unthrottle"
    DO "set", key: "throttled", value: false
    return true
```

Law 8.5: `root` law

```
Name: root
LawScript: CoffeeScript

UPON "adopted", ->
  DO "set", key: "root_turn"
  return true

UPON "arrived", ->
  DO "deliver"
  return true

UPON "sent", ->
  if CS("root_turn")
    DO "set", key: "root_turn", value: false
    DO "forward", message: "root"
    DO "forward", message: @message
  else
    DO "set", key: "root_turn"
    DO op for op in DELEGATE()
  return true
```

Law 8.6: `dummy-hierarchy` law

```
Name: dummy-hierarchy
LawScript: CoffeeScript
Portal: root, http://192.168.1.100/root.law
Refines: root

UPON "sent", ->
  DO "forward", message: "sub"
  DO "forward", message: @message
  return true
```

8.2 Performance Evaluation of the Discovery Mechanism

This set of experiments are for evaluating the performance of the Discovery mechanism of our prototype, i.e., the time it needs to discover a failed controller after the failure happens.

The discovery of a failed controller after its failure involves two stages:

1. The Ledger Writer records the events/operations of the controller reflecting its abnormal behavior on the Ledger as Ledger Entries.

2. The Inspector inspects these Ledger Entries and discovers the failed controller.

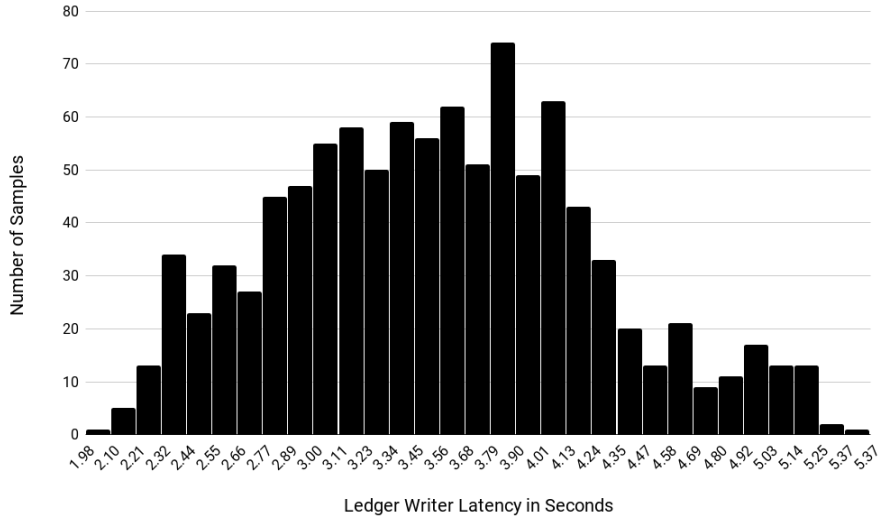
Therefore, the performance of the Discovery mechanism depends on the performance of both Stage 1 and Stage 2, i.e., the performance of the Ledger Writer and the Inspector.

8.2.1 Performance Evaluation of the Ledger Writer

We first evaluated the performance of the Ledger Writer of our prototype, i.e., the time it takes the Ledger Writer to record an event/operation on the Ledger after it occurs at or is executed by a controller. Our evaluation was conducted on 1 CP Node running as a VirtualBox VM with eight 3.0 GHz virtual CPU cores, 8GB RAM, 7200 RPM HDD, and Debian OS. In order to minimize the uncertainty brought by network latency, the Ledger was stored on the same physical machine.

We spawned 2 controllers following Law 8.1 (`trivial` law), each of which would send a message of 1K bytes to the other one every second. This led to 8 events/operations per second: 2 *sent* events, 2 *forward* operations, 2 *arrived* events, and 2 *deliver* operations.

Figure 8.1: Distribution of the Latency of Ledger Writer



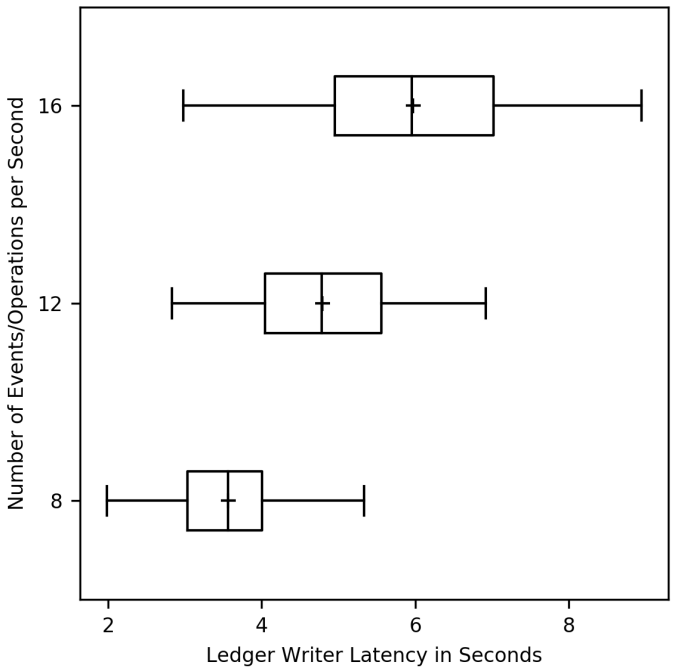
During the experiment, we recorded 1000 samples of the time it took the Ledger Writer to record an event/operation on the Ledger after it occurred at or was executed by any one of the controllers. Figure 8.1 is a histogram showing the distribution of these results, in which the x-axis is the latency of the Ledger Writer in seconds, the y-axis is the number of samples. The results show that:

- The average latency is 3.56 seconds
- The median latency is 3.54 seconds.
- The minimal latency is 1.98 seconds.
- The maximum latency is 5.37 seconds.
- The standard deviation is 0.70.

We also evaluated the impact of event/operation frequency on the performance of the Ledger Writer. We spawned 4 controllers following Law 8.1 (`trivial` law) on the CP Node, then conducted 3 experiments with respectively 2, 3, and 4 of the controllers conducted interactive activities every second each of which would send a message of 1K bytes to a random controller on the CP Node. This led to 8, 12, and 16 events/operations per second respectively. During each of the experiments, we recorded 1000 samples of the time it took

the Ledger Writer to record an event/operation on the Ledger after it occurred at or was executed by any one of the controllers. Figure 8.2 is a box plot showing the distributions of these results, in which the left and right borders of each box show the first and third quartile respectively, the line and cross inside the box show the median and mean value respectively, the leftmost and rightmost bars show the minimum and maximum value respectively. The results show that with the same number of controllers, the frequency of events/operations has an impact on the performance of the Ledger Writer: the increase of 1 event/operation per second with the payload of 1K bytes will increase the average latency of the Ledger Writer by around 0.3 seconds.

Figure 8.2: Impact of Event/Operation Frequency



Frequency	**Mean**	**Median**	**Min**	**Max**	**Standard Deviation**
8	3.57	3.56	1.99	5.34	0.73
12	4.79	4.78	2.83	6.92	0.92
16	5.97	5.96	2.98	8.94	1.25

Furthermore, we evaluated the impact of the number of controllers on the performance of the Ledger Writer. In addition to the previous experiments with 8 events/operations per second and 2 and 4 controllers, we conducted a similar experiment with 3 controllers on the CP Node while keeping the event/operation frequency and the payload size. We did this by

making only two randomly chosen controllers conduct interactive activities every second, each of which would send a message of 1K bytes to a random controller on the CP Node. During the experiment, we recorded 1000 samples of the time it took the Ledger Writer to record an event/operation on the Ledger after it occurred at or was executed by any one of the controllers. Table 8.1 shows the distributions of these results, along with the results of the previous experiments with 2 and 4 controllers. The results show that under the same event/operation frequency, the number of controllers does not have a significant impact on the performance of the Ledger Writer.

Table 8.1: Impact of the Number of Controllers

#	Mean	Median	Min	Max	Standard Deviation
2	3.56	3.54	1.98	5.37	0.70
3	3.53	3.51	2.04	5.37	0.73
4	3.57	3.56	1.99	5.34	0.73

8.2.2 Performance Evaluation of the Inspector

After the evaluation of the Ledger Writer, we evaluated the performance of the Inspector of our prototype, i.e., the time it takes the Inspector to discover the failed controller after the corresponding Ledger Entries appear on the Ledger.

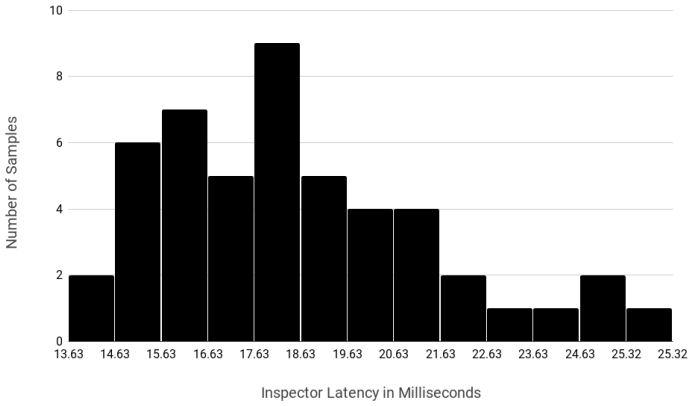
Our evaluation was conducted on a VirtualBox VM with eight 3.0 GHz virtual CPU cores, 8GB RAM, 7200 RPM HDD, and Debian OS. On the VM, we ran an Inspector for Law 8.1 (`trivial` law) that kept inspecting the Ledger which was stored on the same physical machine.

We conducted 50 experiments in each of which we spawned a controller following Law 8.1 (`trivial` law) in the beginning, then started to silently follow Law 8.2 (`trivial-rogue` law) after 30 seconds, which should be regarded as failed as it violated Law 8.1. During each experiment, we recorded the time it took the Inspector to discover the failed controller after the corresponding Ledger Entries appeared on the Ledger. Figure 8.3 is a histogram showing the distribution of the results, in which:

- The average latency is 18.50 milliseconds.

- The median is 18.11 milliseconds.
- The minimal latency is 13.63 milliseconds.
- The maximum latency is 25.31 milliseconds.
- The standard deviation is 2.93.

Figure 8.3: Distribution of the Latency of Inspector



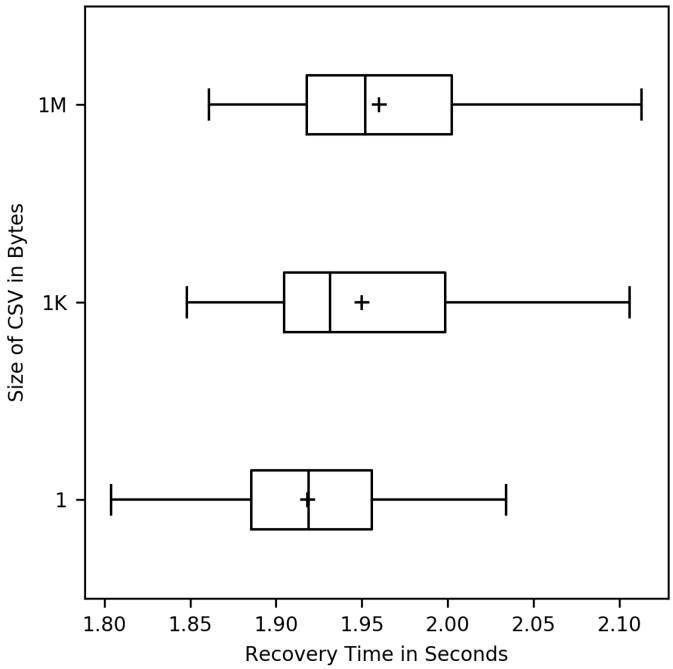
8.3 Performance Evaluation of the Recovery Mechanism

This set of experiments are for evaluating the performance of the Recovery mechanism of our prototype, i.e., the time it needs to recover a failed controller.

We have measured the recovery time for different sizes of CSV on 1 CP Node running as a VirtualBox VM with eight 3.0 GHz virtual CPU cores, 8GB RAM, 7200 RPM HDD, and Debian OS. The CSV sizes we considered include 1 byte, 1K bytes, and 1M bytes. For each CSV size, we conducted 50 experiments, in each of which we recovered a failed controller with a CSV of that size. Figure 8.4 shows the results of our experiments.

The results show that a failed controller can be normally recovered in around 2 seconds, and the size of CSV has an insignificant impact on the performance of the Recovery mechanism.

Figure 8.4: Time to Recover a Controller



Size of CSV	Mean	Median	Min	Max	Standard Deviation
1	1.92	1.92	1.80	2.03	0.06
1K	1.95	1.93	1.85	2.11	0.06
1M	1.96	1.95	1.86	2.11	0.06

8.4 Discussion on the Evaluation

The evaluation results show that in our prototype:

- The D&R mechanism works correctly.
- Under the frequency of 8 events/operations per second with the payload of 1K bytes, the Ledger Writer can record an event/operation on the Ledger after it occurs at or is executed by a controller in averagely 3.56 seconds.
- The increase of 1 event/operation per second with the payload of 1K bytes will increase the average latency of the Ledger Writer by around 0.3 seconds.
- The number of controllers does not have a significant impact on the performance of the Ledger Writer.

- The Inspector can discover a failed controller after the corresponding Ledger Entries appear on the Ledger in averagely 18.50 milliseconds.
- After being discovered, a failed controller can be recovered in around 2 seconds. The size of CSV does not have a significant impact.

It is noteworthy that although in our prototype the D&R mechanism works correctly and the performance of the Recovery mechanism is reasonably decent and stable, the performance of the Discovery mechanism is relatively inferior, and the main bottleneck of the Discovery mechanism is the Ledger Writer.

As mentioned in the beginning of this chapter, our experiments were conducted with weak hardware; in particular, the hard drive we used for the performance evaluation is very slow and the CPU we used has mediocre single-core frequency. This has a significant impact on the performance of the Ledger Writer, as the way it works depends on the capture and analysis of network I/O, which inherently incur high pressure on the CPU and the hard drive. In addition, the Ledger Writer frequently writes to the Ledger; therefore, the write speed of the Ledger also has a significant impact on the performance of the Ledger Writer. With better hardware, especially faster hard drive such as Solid-State Drive (SSD), the performance of the Ledger Writer can be significantly improved.

Moreover, although our prototype is functionally complete, it is not optimized in terms of performance. In particular, the Ledger Writer of our prototype is written in Python, an interpreted scripting language with significantly lower performance than compiled languages such as C and C++. Meanwhile, the performance of various parts of the Ledger Writer can be significantly improved with technical approaches that are more performance-oriented.

An industry-level implementation of Cop is expected to have sufficient performance optimization and be deployed on a more powerful hardware platform. We are optimistic that in an industry-level implementation of Cop, the performance of the Discovery mechanism can be improved by at least an order of magnitude.

Chapter 9

Future Directions

The research on Cop can be continued in several directions. This chapter outlines four of the most important ones that I plan to pursue in the future.

The first direction is to minimize the damage already inflicted by failed controllers. As discussed in Section 5.2.2, Cop can handle the Minus Case largely, and the Plus Case to a certain extent; however, it does not have a systematical method for handling the Plus Case. Therefore, such a systematical method deserves broader and deeper exploration in the future.

The second direction is to optimize the performance of the D&R mechanism. Since both the Ledger Writer and the Inspector can benefit from parallelization, it is worth exploring to accelerate them with parallel computing platforms and frameworks such as CUDA [44] and OpenCL [53]. In addition, as mentioned in Section 8.4, the performance of the Ledger Writer may be improved in several ways, which are worth exploring: it may be improved with faster storage devices such as SSD, storage architectures with better performance such as RAID-0 [45], or file systems with better performance; it may also be improved with programming languages with higher performance or technical approaches that are more performance-oriented. Finally, since the purpose of each component of the D&R mechanism is quite specific, it may be possible to design specialized system infrastructure for each component in order to optimize the performance.

The third direction is to further enhance the trustworthiness and security of Cop's TCB. The trustworthiness of the Inspectors and CP may be enhanced with traditional trusted computing technologies such as TPM, and the security of the Ledgers may be enhanced with replication mechanisms. Moreover, it is worth exploring to move the Ledger Writer out of the CP Node, so that even if the CP Node is compromised, the Ledger Writer will

still produce correct Ledger Entries.

The fourth direction is to explore the applications of Cop. Cop can enforce protocols in single-community MAS, multi-community MAS, or more complex systems such as Governed Distributed Systems (GDS) [38]. Many open questions in such systems can potentially be solved by Cop.

Chapter 10

Conclusion

This dissertation introduces Cop, a broad-spectrum, dependable, and secure protocol enforcement mechanism for Multi-Agent Systems. Cop features low latency, high scalability, and regulated interoperability, making it applicable to time-critical systems, large-scale systems, complex systems, and systems-of-systems. Cop is arguably superior to the currently popular Blockchain-based Smart Contract mechanisms in terms of its applicability to a wide range of systems, as the latter inherently has high latency and low scalability, and lacks interoperability.

Cop is based on Law-Governed Interactions (LGI), with which it carries out the protocol enforcement. LGI is a mechanism for a community of distributed actors to interact under an explicit and enforced protocol called the law of the community, which is enforced by mutually independent controllers in a strictly decentralized way. While LGI can enforce protocols with low latency, high scalability, and regulated interoperability, it has an Achilles' Heel which may degrade its dependability and security: it provides no general mechanism for handling failed controllers; in particular, it provides no general mechanism for discovering and recovering failed controllers.

Cop removes the Achilles' Heel of LGI with a protective mechanism called Discovery & Recovery (D&R), which can promptly discover failed controllers right after the failure occurs, and can immediately recover the failed controllers right after discovering them and assist with the reparation of the damage they may have inflicted.

In Cop, controllers are hosted by a component called Controller Provider (CP), which consists of a cluster of servers called CP Nodes and a component for managing the cluster called CP Manager. Each CP Node hosts multiple generic controllers which may serve different communities, and each controller is isolated in a Linux container, preventing it

from interfering with other cotenant controllers or components on the CP Node when it fails.

For each community, Cop records the interactive activities of all controllers serving the community in a structure called Ledger which consists of time-ordered Ledger Entries, each of which represents an event occurred at a controller or an operation that a controller executed. Since the controllers may fail, the production of Ledger Entries cannot rely on the controllers' self-reporting. In Cop, Ledger Entries are produced in a reliable way: on each CP Node, a component called Ledger Writer captures the network I/O of the controllers hosted by the CP Node, with which it can accurately infer the events occurred at the controllers and the operations the controllers have executed, then produce the corresponding Ledger Entries and write to the corresponding Ledgers.

Each community has an Inspector which systematically and continuously inspects the community's Ledger to discover failed controllers. The inspection is highly efficient and scalable thanks to the local nature of LGI laws and their enforcement; therefore, the Inspector can promptly discover any controller that fails to comply with the community's law right after the failure occurs. Immediately after the discovery of a failed controller, the Inspector will initiate its recovery, which will be actuated by CP in a highly efficient and scalable way thanks to the local and generic nature of controllers. In addition, the Inspector also generates a diagnosis report according to the Ledger for the failed controller describing its incorrect behavior, which can assist with the reparation of the damage it may have inflicted. The D&R mechanism can effectively handle isolated communities as well as interoperating communities.

To ensure the correctness of the D&R mechanism, Cop has several requirements on LGI's implementation. We have made several enhancements to LGI's implementation to address all these requirements.

We have developed a functionally complete prototype of Cop, and conducted several experiments with it. Through the experiments, we verified the correctness of the D&R mechanism in our prototype, and studied its performance to an extent that is meaningful for a prototype and weak hardware.

Appendix A

Introduction to LawScript

LawScript is a DSL for writing LGI laws designed by the dissertation author and is fully supported in Moses-2 introduced in Chapter 7. The example laws in this dissertation are all written in LawScript. To help the readers better understand the examples, this chapter provides an introduction to LawScript.

LawScript supports *dialects*, i.e., variants based on the syntaxes of different languages. Any JVM language can be developed into a LawScript dialect in principle. Currently, two dialects are implemented, which are based on JavaScript and CoffeeScript respectively. This chapter focuses on the CoffeeScript-based dialect since all example laws in the dissertation are written in it.

CoffeeScript is a scripting language semantically equivalent to JavaScript but with better readability and flexibility. This chapter does not cover the details of CoffeeScript since there are many decent introductions such as [33]. Before reading on, the reader is suggested to get familiar with CoffeeScript first.

A.1 The Structure of LawScript Code

A LawScript code is a plain text file written in LawScript, conventionally with the extension name of "`.law`". A LawScript code consists of 3 parts:

1. Preamble.
2. Definition of constants and functions.
3. ECA blocks.

Snippet A.1: Preamble with all elements

```
Name: State
LawScript: CoffeeScript
Portal: Federal, http://192.168.1.100/federal.law
Portal: Constitution, http://192.168.1.100/constitution.law
Refines: Federal

...the rest of the law...
```

A.1.1 Preamble

Preamble is the first and mandatory part of a LawScript code, it includes:

- The law's name. It should be specified on the first line of Preamble, in the format of: `Name: $n`, in which `$n` is the law's name.
- The LawScript dialect being used. It should be specified on the second line of Preamble, in the format of: `LawScript: $d`, in which `$d` is the dialect's name. To use CoffeeScript dialect, `$d` should be `CoffeeScript`.
- The pointers to other involved laws, called *portals*. A law may specify zero or more portals, each of which is specified in the format of: `Portal: $n, $l`, in which `$n` is the portal's name, `$l` is the URL of the law. Every law has a special portal named `this_law` pointing to itself.
- The parent law's portal name if the law is in a Conformance Hierarchy, in the format of: `Refines: $p`, in which `$p` is the parent law's portal name, which should be one of the portal names defined above.
- An empty line marking the end of Preamble. Everything after the empty line will not be treated as part of Preamble.

Snippet A.1 shows the Preamble of a LawScript code with all elements, in which the law name is `State`, dialect is CoffeeScript, two portals `Federal` and `Constitution` are defined, and the parent law is the law pointed by the portal `Federal`.

Snippet A.2: A simple ECA block

```
UPON "arrived", ->
  if @message is "foo" and CS("bar") is true
    DO "deliver"
    return true
```

A.1.2 Definition of Constants and Functions

After Preamble is the body of the LawScript code. Constants and helper functions can be defined at the beginning of the body. The names of constants and functions *must not* be the following reserved words: `MOSES`, `UPON`, `DO`, `CS`, `DELEGATE`, `CONFORMS`, `RANDOM`.

A.1.3 ECA Blocks

ECA blocks are the main part of a LawScript code, each of which specifies when an event occurs, under a specific condition, what operations should be executed sequentially and atomically. An ECA block has the structure of: `UPON "$t", $h`, in which `$t` is the event type that this ECA block is to process, and `$h` specifies the event handler: a CoffeeScript function that processes the event, which can be in the form of either the definition of an anonymous function or the name of a function defined before. Note that `$t` is in underscore case, i.e., for an *obligationDue* event, `$t` equals to `obligation_due`.

LawScript supports *handler fallthrough*: in a LawScript code, there may be multiple ECA blocks processing the same event type; when the event occurs, these ECA blocks will be executed in order until one of them executes `return true`, which prevents further ECA blocks for the event type from being executed.

LawScript provides some salt and pepper that can be used in event handlers, including `DO, CS, DELEGATE, CONFORMS, RANDOM`. The following sections will elaborate on them.

Snippet A.2 shows a simple ECA block, with the meaning of: when an *arrived* event occurs, if its `message` argument equals to `foo` and the control state entry `bar` equals to `true`, then it mandates a *deliver* operation with default argument values, and stops further ECA blocks for *arrived* events from being executed.

A.2 Accessing Event Arguments

As mentioned in Section 2.1, an event may carry several arguments. The value of an argument may be in the form of any native CoffeeScript data structure, including number, string, boolean, null, array, and object. In an event handler, the value of an event argument can be accessed with `@$k`, in which `$k` is the argument's name. The names of all event arguments are listed in Table A.1. If the event does not have an argument with the specified name, `undefined` will be returned.

For example, the `message` argument of an *arrived* event can be returned with `@message`.

In addition to its own arguments, every event has 2 special arguments:

- `self`: the name of the controller at which the event occurs.
- `time`: the timestamp when the event occurs.

Table A.1 lists the event arguments, including their names, value types, and descriptions. The events without arguments are omitted.

Table A.1: Event Arguments

Event	Argument	Type	Description
adopted	`arguments`	object	adoption arguments
	`issuer`	object	issuer of actor's certificate
	`subject`	object	subject of actor's certificate
arrived	`message`	any	the arrived message
	`sender`	string	sender name
	`law`	string	the portal name of the sender's law
exception	`type`	string	exception type
	`message`	object	operation that caused the exception
obligationDue	`name`	string	obligation name
	`arguments`	object	obligation arguments
sent	`message`	any	the message to send
	`receiver`	string	receiver name
submitted	`message`	any	the arrived message
	`address`	string	sender address
	`port`	number	sender port
	`issuer`	object	issuer of the sender's certificate
	`subject`	object	subject of the sender's certificate

Snippet A.3: Alternative way of mandating an operation

```
op = {operation_type: "set", key: "foo", value: "bar"}
DO op
```

A.3 Mandating Operations

An event handler can mandate the operations that are to be executed sequentially and atomically for the event when the conditions are satisfied. An operation can be mandated with `DO "$t", $a`, in which `$t` is the operation's type, and `$a` represents the arguments of the operation in the form of key-value pairs, in each of which the key is the argument's name, and the value is the expected value which may be in the form of any native CoffeeScript data structure, including number, string, boolean, null, array, and object. `$a` can be in the form of either the definition of a CoffeeScript object or the name of a CoffeeScript object defined before. The names of all operation arguments are listed in Table A.2. Note that `$t` is in underscore case, i.e., for an *imposeObligation* operation, `$t` equals to `impose_obligation`.

For example, a *set* operation with arguments `key = "foo"` and `value = "bar"` can be mandated with

```
DO "set", key: "foo", value: "bar"
```

An operation can also be mandated with `DO $o`, in which `$o` is a CoffeeScript object representing an operation, in the form of: `{"operation_type": $t, "$k0": $v0, "$k1": $v1, ...}`, in which `$t` is the operation's type, each `$k` is the name of an argument, each `$v` is the corresponding value. This method is particularly handy in constructing Conformance Hierarchy, which will be discussed in Section A.5. Snippet A.3 shows an example of this method, which mandates the same operation as the example above.

Table A.2 lists the operation arguments, including their names, value types, descriptions, and default values. Arguments without default values are mandatory. The operations without arguments are omitted.

Table A.2: Operation Arguments

Operation	Argument	Type	Description	Default value
deliver	`sender`	string	the message to send	`sender` of the *arrived* event
	`message`	any	message sender	`message` of the *arrived* event
forward	`sender`	string	message sender	`self`
	`receiver`	string	message receiver	`receiver` of the *sent* event
	`law`	string	the portal name of the receiver's law	`this_law`
	`message`	any	the message to send	`message` of the *sent* event
imposeObligation	`name`	string	obligation name	
	`arguments`	any	obligation arguments	null
	`delay`	number	obligation delay in ms	
release	`address`	string	receiver address	
	`port`	number	receiver port	
	`message`	any	the message to send	
	`use_tls`	boolean	use TLS or not	false
repealObligation	`name`	string	obligation name	
set	`key`	string	control state entry name	
	`value`	any	control state entry value	true
unset	`key`	string	control state entry name	

A.4 Accessing Control State

Each control state entry has a name and a value, the value can be any CoffeeScript object. The value of a control state entry can be returned with `CS("$n")`, in which `$n` is the name of the control state entry. If the control state entry with the specified name does not exist, `undefined` will be returned.

For example, `CS("foo")` returns the value of the control state entry named `foo`.

The creation, modification, and removal of control state entries are done by *set*/*unset* operations, which will be executed together with other operations mandated for the current event atomically; therefore, reading a control state entry right after creating, modifying, or removing it in the same event handler will only yield its old value instead of the new one. For example, consider Snippet A.4. The statement `DO "forward"` will not be executed, because `CS("foo")` still returns `undefined`. Snippet A.5 shows the proper way to access the new control state entry value right after creating or modifying it in the same event handler.

Snippet A.4: Example A of control state reading

```
DO "set", key: "foo", value: "bar"
if CS("foo") is "bar"
  DO "forward"
```

Snippet A.5: Example B of control state reading

```
foo = "bar"
DO "set", key: "foo", value: foo
if foo is "bar"
  DO "forward"
```

A.5 Constructing Conformance Hierarchy

To construct a Conformance Hierarchy, a law should have the following abilities:

- Specifying the parent law.
- Conducting dynamic consultation.
- Conducting conformance check.

As introduced in Section A.1.1, in a LawScript code the parent law can be specified in Preamble part.

Dynamic consultation in LawScript can be conducted in each event handler with `DELEGATE()`, which returns the sequence of operations mandated by the subordinate law for the event. Each operation in the sequence is represented as a CoffeeScript object with the structure of: `{"operation_type": $t, "$k0": $v0, "$k1": $v1, ...}`, in which `$t` is the operation's type, each `$k` is the name of an argument, each `$v` is the corresponding value.

Snippet A.6: Trivial dynamic consultation

```
for op in DELEGATE():
  DO op
```

Snippet A.7: Dynamic consultation with condition

```
for op in DELEGATE():
  if op.operation_type is "set"
    DO op
```

Snippet A.8: Dynamic consultation with revision

```
for op in DELEGATE():
  if op.operation_type is "forward"
    op.message = "sub:#{op.message}"
    DO op
```

Snippet A.6 shows a trivial dynamic consultation, which will mandate any operations the subordinate law mandates. Snippet A.7 shows a dynamic consultation with condition, which will only mandate all *set* operations the subordinate law mandates. Snippet A.8 shows a dynamic consultation with revision, which will mandate all *forward* operations the subordinate law mandates after prepending the message with "`sub:`".

Conformance check can only be conducted in *arrived* event handlers, in the format of: `CONFORMS($p)`, in which `$p` is a portal name. It returns true if the event type is *arrived* and the law of the sender conforms to the law pointed by the portal `$p`. Otherwise it returns false.

A.6 Generating Random Numbers

To write a non-deterministic law, one can use `RANDOM()` to generate a 32-bit random number.

Appendix B

Protocols of Moses-2

B.1 Message Encoding

In Moses-2, there are two types of messages:

1. Moses Message, or M-message for short. The vast majority of the messages sent and received by Moses-2 controllers are of this type. An M-message has the following elements in order:

   (a) A byte `0x77` (M), marking this is an M-message.

   (b) A byte indicating the type of the M-message. Table B.1 lists the possible values of the byte and the types they represent.

   (c) The message body: a data structure encoded in Protocol Buffers [19] format.

2. JSON [11] string, whose first byte may be:

   - `0x5B` (`[`, initial of an array)
   - `0x7B` ({, initial of an object)
   - `0x22` (`"`, initial of a string)
   - `0x30` - `0x39` (0-9, initial of a number)
   - `0x2D` (`-`, initial of a negative number)
   - `0x6E` (`n`, initial of `null`)

A correct Moses-2 controller will discard network input streams whose payload does not start with `0x77`, `0x5B`, `0x7B`, `0x22`, `0x30`, `0x2D`, `0x6E`. On the other hand, a correct Moses-2 controller will never initiate a network output stream whose payload does not start with these bytes.

Table B.1: M-Message Types

`0x03`: *adoptRequest*	`0x04`: *adoptResponse*
`0x05`: *deliverMessage*	`0x06`: *disconnectRequest*
`0x07`: *disconnectResponse*	`0x08`: *forwardMessage*
`0x09`: *handshakeMessage*	`0x0A`: *quitMessage*
`0x0B`: *sendRequest*	`0x0C`: *sendResponse*
`0x0D`: *imposeObligationRequest*	`0x0E`: *imposeObligationResponse*
`0x0F`: *repealObligationRequest*	`0x10`: *repealObligationResponse*
`0x11`: *obligationDueMessage*	

These two types of messages are very convenient for the Ledger Writer to accurately and efficiently decode from network stream payload. First, the Ledger Writer checks the first byte. If it is one of `0x5B`, `0x7B`, `0x22`, `0x30`, `0x2D`, `0x6E`, then the rest of the stream payload can be safely parsed as a JSON string; if it is `0x77`, then the stream payload should be parsed as M-messages; otherwise, the stream does not contain any valid message. If the stream payload contains M-messages, then it should be parsed as follows:

1. Use *cursor* to denote the ordinal number of the byte that is about to be read in the stream payload. Initially, the cursor points to the first byte of the stream payload.

2. Check the byte pointed by the cursor, if it is not `0x77`, it means the rest of the stream payload is corrupted, then go to Step 6.

3. Move the cursor to the next byte and decode the type of the M-message from the byte. If no valid type can be decoded, it means the rest of the stream payload is corrupted, then go to Step 6.

4. Move the cursor to the next byte and decode the body of the M-message. See [20] for the details of Protocol Buffers encoding. As discussed in [34], data structures encoded in Protocol Buffers can be very efficiently decoded from bytes. If no valid body can be decoded, it means the rest of the stream payload is corrupted, then go to Step 6.

5. Now one M-message has been decoded, and the cursor points to the last byte of its body. If there are still bytes in the stream payload, it means there may be other M-messages, so move the cursor to the next byte and go to Step 2 to decode the next M-message. Otherwise, go to Step 6.

6. Stop parsing and return the already decoded M-messages.

B.2 Events

This section introduces the protocols for triggering events on a Moses-2 controller. The discussion will use the event argument names listed in Table A.1.

B.2.1 *adopted*

To adopt a controller, the actor must have the correct controller password k. When a controller receives an M-message $adoptRequest(k', p, a)$ via a TLS stream, if the controller is not yet adopted and $k' = k$, then the controller records the sender's address as A_a, p as A_p, and the sender's certificate as A_c; then an *adopted* event will be triggered, whose `arguments` argument equals to a, `issuer` and `subject` arguments equal to the issuer and subject information of A_c respectively; finally an M-message $adoptResponse()$ will be sent back to the actor.

B.2.2 *arrived*

When a controller receives an M-message $handshakeMessage(h)$ via a TLS stream, and the sender's certificate is signed by a trusted controller CA, then it will respond with a $handshakeMessage(h')$ message, in which h' is the controller's law hash; if the sender then responds with an M-message $forwardMessage(m, s)$, it means the handshake is successful, then an *arrived* event will be triggered, whose `message` argument equals to m, `sender` argument equals to s, `law` argument equals to the name of the portal whose law hash equals to h.

B.2.3 *sent*

To trigger a *sent* event on a controller, the actor must have the correct controller password k. When a controller receives an M-message $sendRequest(k', m, r)$ via a TLS stream, if $k' = k$, the sender's address equals to A_a and the sender's certificate equals to A_c, then

a *sent* event will be triggered, whose `message` argument equals to m, `receiver` argument equals to r; finally an M-message *sendResponse*() will be sent back to the actor.

B.2.4 *submitted*

When a controller receives a JSON string, a *submitted* event will be triggered, whose `message` argument equals to the object decoded from the JSON string, `address` and `port` arguments equal to the sender's address and port respectively, `issuer` and `subject` arguments equal to the issuer and subject information of the sender's certificate respectively. If the JSON string is not sent through TLS connection, `issuer` and `subject` arguments equal to `null`.

B.2.5 *obligationDue*

When a controller receives an M-message $obligationDueMessage(n, a)$ via a TLS stream, if the sender's address and certificate equal to those of the Obligation Registry, then an *obligationDue* event will be triggered, whose `name` argument equals to n, `arguments` argument equals to a.

B.2.6 *disconnected*

To trigger a *disconnected* event on a controller, the actor must have the correct controller password k. When a controller receives an M-message $disconnectedRequest(k')$ via a TLS stream, if the controller is not disconnected, $k' = k$, the sender's address equals to A_a, and the sender's certificate equals to A_c, then a *disconnected* event will be triggered, and an M-message *disconnectedResponse*() will be sent back to the actor. If the controller is not disconnected, a failed *deliver* operation, which will be described in Section B.3.2, will also trigger a *disconnected* event.

B.2.7 *reconnected*

To trigger a *reconnected* event on a controller, the actor must have the correct controller password k. When a controller receives an M-message $adoptRequest(k', p, a)$ or $sendRequest(k', m, r)$ via a TLS stream, if the controller is adopted and disconnected,

$k' = k$, the sender's address equals to A_a, and the sender's certificate equals to A_c, then a *reconnected* event will be triggered. If the M-message is $sendRequest(k', m, r)$, then it will also trigger the *sent* event described in Section B.2.3

B.2.8 *exception*

exception events will be discussed in Section B.3.

B.3 Operations

This section introduces the protocols for a Moses-2 controller to execute operations. The discussion will use the operation argument names listed in Table A.2.

B.3.1 *forward*

When a controller executes a *forward* operation o, it sends an M-message $handshakeMessage(h)$ to the controller specified in o's `receiver` argument via a TLS stream, in which h is the controller's law hash; if the receiver replies with a $handshakeMessage(h')$, its certificate is signed by a trusted controller CA, and h' equals to the law hash of the portal whose name equals to o's `law` argument, then it means the handshake is successful; then the controller sends an M-message $forwardMessage(m, s)$ to the receiver, in which m equals to o's `message` argument, s equals to o's `sender` argument. If the handshake fails, then an *exception* event will be triggered, whose `type` argument equals to `HANDSHAKE_EXCEPTION` and `operation` argument equals to o. If $forwardMessage(m, s)$ fails to be sent, then an *exception* event will be triggered, whose `type` argument equals to `FORWARD_EXCEPTION` and `operation` argument equals to o.

B.3.2 *deliver*

When a controller executes a *deliver* operation o, it sends an M-message $deliverMessage(m, s)$ to the actor at address A_a port A_p via a TLS stream, in which m equals to o's `message` argument, s equals to o's `sender` argument. If o fails, then an *exception* event will be triggered, whose `type` argument equals to `DELIVER_EXCEPTION` and

`operation` argument equals to o. If the controller is not disconnected, a failed o will also trigger a *disconnected* event, as discussed in Section B.2.6.

B.3.3 *release*

When a controller executes a *release* operation o, it sends a JSON string representing o's `message` argument to the address and port specified by o's `address` and `port` arguments without TLS; if o's `use_tls` argument is true then it will use TLS. If o fails, then an *exception* event will be triggered, whose `type` argument equals to `RELEASE_EXCEPTION` and `operation` argument equals to o.

B.3.4 *imposeObligation*

When a controller executes an *imposeObligation* operation o, it sends an M-message $imposeObligationRequest(n, a, d)$ to the Obligation Registry via a TLS stream, in which n equals to o's `name` argument, a equals to o's `arguments` argument, d equals to o's `delay` argument. If the Obligation Registry responds with an M-message $imposeObligationResponse()$ without errors, then o is successful; otherwise, an *exception* event will be triggered, whose `type` argument equals to `IMPOSE_OBLIGATION_EXCEPTION` and `operation` argument equals to o.

B.3.5 *repealObligation*

When a controller executes a *repealObligation* operation o, it sends an M-message $repealObligationRequest(n)$ to the Obligation Registry via a TLS stream, in which n equals to o's `name` argument. If the Obligation Registry responds with an M-message $repealObligationResponse()$ without errors, then o is successful; otherwise, an *exception* event will be triggered, whose `type` argument equals to `REPEAL_OBLIGATION_EXCEPTION` and `operation` argument equals to o.

B.3.6 *quit*

When a controller executes a *quit* operation, it sends an M-message $quitMessage()$ to the actor at address A_a port A_p via a TLS stream.

B.3.7 *set/unset*

Operations of these types will not cause interactions.

References

[1] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al. Hyperledger Fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, page 30. ACM, 2018.

[2] X. Ao and N. H. Minsky. Flexible regulation of distributed coalitions. In *European Symposium on Research in Computer Security*, pages 39–60. Springer, 2003.

[3] X. Ao, N. H. Minsky, T. Nguyen, and V. Ungureanu. Law-governed communities over the Internet. In *Proceedings of the Fourth International Conference on Coordination Models and Languages*, pages 133–147, 2000.

[4] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1st edition, 2014.

[5] W. Arthur and D. Challener. *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. Apress, 1st edition, 2015.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177. ACM, 2003.

[7] S. Blakstad and R. Allen. *FinTech Revolution: Universal Inclusion in the New Financial Ecosystem*. Springer, 1st edition, 2018.

[8] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pages 173–186. USENIX Association, 1999.

[9] M. Castro and B. Liskov. Proactive recovery in a Byzantine-fault-tolerant system. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation*, volume 4, page 19. USENIX Association, 2000.

[10] L. Chappell and G. Combs. *Wireshark Network Analysis: The Official Wireshark Certified Network Analyst Study Guide*. Chappell University, 2nd edition, 2012.

[11] D. Crockford. The application/json media type for JavaScript Object Notation (JSON). *RFC*, 4627:1–9, 2006.

[12] J. Daemen and V. Rijmen. *The Design of Rijndael: AES-the Advanced Encryption Standard*. Springer, 1st edition, 2013.

[13] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) protocol version 1.2. *RFC*, 5246:1–103, 2008.

[14] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.

[15] R. Dudheria, W. Trappe, and N. H. Minsky. Coordination and control in mobile ubiquitous computing applications using law governed interaction. In *Proceedings of the Fourth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies*, pages 247–256. IARIA, 2010.

[16] B. Eckel. *Thinking in Java*. Prentice Hall Professional, 4th edition, 2005.

[17] K. R. Fall and W. R. Stevens. *TCP/IP Illustrated, Volume 1: the Protocols*. Addison-Wesley Professional, 2nd edition, 2011.

[18] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and Linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 171–172. IEEE, March 2015.

[19] Google. Protocol Buffers. *URL https://developers.google.com/protocol-buffers/*, last accessed on 2019-01-17.

[20] Google. Protocol Buffers: Encoding. *URL https://developers.google.com/protocol-buffers/docs/encoding/*, last accessed on 2019-01-17.

[21] Google. Protocol Buffers: Language guide. *URL https://developers.google.com/protocol-buffers/docs/proto3/*, last accessed on 2019-01-17.

[22] M. Ionescu, N. H. Minsky, and T. Nguyen. Enforcement of communal policies for peer-to-peer systems. In *Proceedings of the Sixth International Conference on Coordination Models and Languages*, pages 152–169, 2004.

[23] V. Jacobson, C. Leres, and S. McCanne. *The tcpdump Manual*. Lawrence Berkeley Laboratory, Berkeley, CA, April 2018. Available at URL https://www.tcpdump.org/manpages/tcpdump.1.html, last accessed on 2019-01-17.

[24] A. M. Joy. Performance comparison between Linux containers and virtual machines. In *2015 International Conference on Advances in Computer Engineering and Applications*, pages 342–346. IEEE, March 2015.

[25] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing protocols for securing group communication. In *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, volume 3, pages 317–326. IEEE, 1998.

[26] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, pages 225–230, 2007.

[27] J. Kwon and E. Buchman. Cosmos: A network of distributed ledgers. *URL https://cosmos.network/whitepaper*, 2016, last accessed on 2019-01-24.

[28] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems (TOCS)*, 10(4):265–310, 1992.

[29] D. H. Lehmer. Mathematical methods in large-scale computing units. In *Proceedings of the Second Symposium on Large Scale Digital Computing Machinery*, pages 141–146. Harvard University Press, 1951.

[30] D. Lezcano, S. Hallyn, S. Graber, et al. Linux containers (LXC). *URL https://linuxcontainers.org/lxc/*, last accessed on 2019-01-17.

[31] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

[32] The Linux Foundation. *Linux Programmer's Manual*, February 2018. Available at URL http://man7.org/linux/man-pages/, last accessed on 2019-01-17.

[33] A. MacCaw. *The Little Book on CoffeeScript*. O'Reilly Media, Inc., 1st edition, 2012.

[34] K. Maeda. Performance evaluation of object serialization libraries in XML, JSON and binary formats. In *2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP)*, pages 177–182. IEEE, May 2012.

[35] P. B. Menage. Adding generic process containers to the Linux kernel. In *Proceedings of the Linux Symposium*, pages 45–57, 2007.

[36] D. Merkel. Docker: Lightweight Linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[37] N. H. Minsky. The imposition of protocols over open distributed systems. *IEEE Transactions on Software Engineering*, 17(2):183–195, 1991.

[38] N. H. Minsky. On the dependability of highly heterogeneous and open distributed systems. *Journal of Software Engineering and Applications*, 11(01):28, 2018.

[39] N. H. Minsky et al. *Law Governed Interaction (LGI): A Distributed Coordination and Control Mechanism*. Rutgers, the State University of New Jersey, New Brunswick, NJ, 2005. Available at URL http://www.moses.rutgers.edu/documentation/manual.pdf, last accessed on 2019-01-17.

[40] N. H. Minsky and T. Murata. On manageability and robustness of open multi-agent systems. In *International Workshop on Software Engineering for Large-Scale Multi-agent Systems*, pages 189–206. Springer, 2003.

[41] N. H. Minsky and V. Ungureanu. Law-Governed Interaction: A coordination and control mechanism for heterogeneous distributed systems. *ACM Transactions on Software Engineering and Methodology*, 9(3):273–305, July 2000.

[42] R. Morabito, J. Kjällman, and M. Komu. Hypervisors vs. lightweight virtualization: a performance comparison. In *2015 IEEE International Conference on Cloud Engineering*, pages 386–393. IEEE, March 2015.

[43] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *URL http://bitcoin.org/bitcoin.pdf*, 2008, last accessed on 2019-01-1.

[44] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, Mar. 2008.

[45] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, volume 17, pages 109–116. ACM, 1988.

[46] G. W. Peters and E. Panayi. Understanding modern banking ledgers through blockchain technologies: Future of transaction processing and smart contracts on the Internet of money. In *Banking Beyond Banks and Money*, pages 239–278. Springer, 2016.

[47] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, 3rd edition, 2007.

[48] M. K. Reiter. The Rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems*, pages 99–110. Springer, 1995.

[49] R. L. Rivest. The MD5 message-digest algorithm. *RFC*, 1321:1–21, 1992.

[50] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[51] J. M. Rushby. Design and verification of secure systems. In *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, pages 12–21. ACM, 1981.

[52] W. R. Stevens, B. Fenner, and A. M. Rudoff. *UNIX Network Programming, Volume 1: the Sockets Networking API*. Addison-Wesley Professional, 3rd edition, 2004.

[53] J. E. Stone, D. Gohara, and G. Shi. OpenCL: a parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66–73, May 2010.

[54] N. Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

[55] S. Thomas and E. Schwartz. A protocol for interledger payments. *URL https://interledger.org/interledger.pdf*, 2015, last accessed on 2019-01-24.

[56] M. Vukolić. Rethinking permissioned blockchains. In *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts*, pages 3–7. ACM, 2017.

[57] D. Wagner, B. Schneier, et al. Analysis of the SSL 3.0 protocol. In *Proceedings of the Second USENIX Workshop on Electronic Commerce*, volume 1, pages 29–40. USENIX Association, 1996.

[58] G. Wood. Ethereum: a secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.

[59] F. Zambonelli, N. R. Jennings, and M. Wooldridge. Developing multiagent systems: The Gaia methodology. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 12(3):317–370, 2003.

[60] W. Zhang, C. Serban, and N. H. Minsky. Establishing global properties of multi-agent systems via local laws. In *International Workshop on Environments for Multi-Agent Systems*, pages 170–183. Springer, 2006.