UNIVERSITÀ DEGLI STUDI DI CAGLIARI

DIPARTIMENTO DI MATEMATICA E INFORMATICA
DOTTORATO DI RICERCA IN MATEMATICA E INFORMATICA
CICLO XXX

PH.D. THESIS

Behavioural contracts: from centralized to decentralized implementations

S.S.D. INF/01

CANDIDATE
Alessandro Sebastian Podda

SUPERVISOR
Prof. Massimo Bartoletti

PHD COORDINATOR
Prof. Giuseppe Rodriguez

Final examination, Academic Year 2016/2017
March, 2018

Acknowledgments

This work is partially supported by Aut. Reg. of Sardinia grant P.I.A. 2013 "NO-MAD". Alessandro Sebastian Podda gratefully acknowledges Sardinia Regional Government for the financial support of her PhD scholarship (P.O.R. Sardegna F.S.E. Operational Programme of the Autonomous Region of Sardinia, European Social Fund 2007-2013 - Axis IV Human Resources, Objective l.3, Line of Activity l.3.1).

Abstract

Developing distributed applications typically requires to integrate new code with legacy third-party services, e.g., e-commerce facilities, maps, etc. These services cannot always be assumed to smoothly collaborate with each other; rather, they live in a "wild" environment where they must compete for resources, and possibly diverge from the expected behaviour if they find it convenient. To overcome these issues, some recent works have proposed to discipline the interaction of mutually distrusting services through *behavioural contracts*.

In the first part of this dissertation, we exploit a theory of timed behavioural contracts to formalise, design and implement a message-oriented middleware in which distributed services can be dynamically composed, and their interaction monitored to detect contract violations. We show that the middleware allows to reduce the complexity of developing distributed applications, by relieving programmers from the need to explicitly deal with the misbehaviour of external services. On the other hand, this middleware introduces a "single point of trust" in the distributed application.

We then explore the possibility that contract-oriented applications safely interact in absence of this trusted entity. To this purpose, the middleware functions are delegated to a network of nodes, that must globally reach a consensus on the decisions to take about the fulfillment of the contracts. We exploit the peer-to-peer network of Bitcoin, a decentralized cryptocurrency introduced in 2009. In particular, we use the Bitcoin blockchain to record tamper-proof execution traces of behavioural contracts, by exploiting the few bytes of metadata that can be carried on standard Bitcoin transactions. Such execution traces form a *subchain* inside the blockchain. Existing approaches either postulate that subchains are always *consistent*, or give weak guarantees about their security (for instance, they are susceptible to Sybil attacks). However, there may exist inconsistent subchains which represent incorrect contract executions.

Thus, in the second part of this thesis, we propose a consensus protocol, based on Proof-of-Stake, that incentivizes nodes to consistently extend the subchain. Finally, we evaluate the security of our protocol, and we show how to exploit it as a basis for implementing behavioural contracts on Bitcoin.

Table of contents

Chapter 1

Introduction

1.1 Motivations

Modern distributed applications are often composed by loosely-coupled services, which can appear and disappear from the network, and can dynamically discover and invoke other services in order to adapt to changing needs and conditions. These services may be under the governance of different providers (possibly competing among each other), and interact through open networks, where competitors can try to exploit their vulnerabilities.

In the setting outlined above, developing trustworthy services and applications can be a quite challenging task: the problem fits within the area of computer security, since we have *adversaries* (in our setting, third-party services), whose exact number and nature is unknown (because of openness and dynamicity). Further, standard analysis techniques from programming languages theory (like e.g., type systems) cannot be applied, since they usually need to inspect the code of the whole application, while under the given assumptions one can only reason about the services under their control.

A possible countermeasure to these issues is to discipline the interaction between services through *contracts*. Contracts are usually descriptions of service behaviour, in terms of, e.g., pre/post-conditions and invariants [57], behavioural types [41], etc. They can be used at static or dynamic time to discover and bind Web services, and to guarantee they interact in a protected manner: when a service does not be-

have as prescribed by its contract, it can be blamed (and punished) for breaching the contract [80]. Although several models and architectures for contract-oriented services have been proposed in the last few years [35, 84, 88], further evidence is needed in order to put this paradigm at work in everyday practice.

These models and architectures usually rely on a *contract broker*, an entity which collects the contracts advertised by the services, helps to establish sessions between compliant services, and monitors the interaction in order to ensure the fulfillment of the contract rules by the parties. A contract broker can be implemented as a *middleware*, i.e. a framework that offers public primitives to the services that want to interact through contracts, ensuring interoperability among applications written in different languages and for different platforms. However, the main limitation of a contract broker is its centralized nature. Since it essentially plays the role of a trusted party, an adversary that takes control of the broker (or manages to exclude it from the network) may compromise the whole system. A possible way to address this issue is to *decentralize* the role of the broker: this means its functions are delegated to all (or some of) the nodes of the network, instead of a single entity. Anyhow, this is not an easy task to achieve, since it requires an effective and safe consensus mechanism among the nodes of the network.

Recently, cryptocurrencies like Bitcoin [73] have pushed forward the concept of decentralization, by ensuring reliable interactions among mutually distrusting nodes in the presence of a large number of colluding adversaries. These cryptocurrencies leverage on a public data structure, called *blockchain*, where they permanently store all the transactions exchanged by nodes, grouped into blocks. Adding new blocks to the blockchain (called *mining*) requires to solve a moderately difficult cryptographic puzzle. The first miner who solves the puzzle earns some virtual currency (some fresh coins for the mined block, and a small fee for each transaction included therein). In Bitcoin, miners must invert a hash function whose complexity is adjusted dynamically in order to make the average time to solve the puzzle $\sim$10 minutes. Instead, removing or modifying existing blocks is computationally unfeasible: roughly, this would require an adversary with more *hashing power* than the rest of all the other nodes. According to the folklore, Bitcoin would resist to attacks unless the adversaries control the majority of total computing power of the Bitcoin network. Even though some vulnerabilities have been reported in the literature (see Section 4.4.3), in practice Bitcoin has worked

surprisingly well so far: indeed, the known successful attacks to Bitcoin are standard hacks or frauds [58], unrelated to the Bitcoin protocol.

The idea of using the Bitcoin blockchain and its consensus protocol as foundations for a decentralized contract-oriented interaction has been explored by several recent works. In particular, Bitcoin offers the possibility to specify some simple contracts in transactions through short programs expressed in a Forth-like scripting language. These contracts are evaluated by the miners before the respective transactions are appended to the blockchain, and often contain clauses which involve the trasfer of some digital currency among nodes: they are commonly referred in literature as *smart contracts* [85]. For instance, [6, 14, 30, 34, 64, 65, 66, 47] propose protocols for secure multiparty computations, fair lotteries, based access control; [48] implements decentralised authorization systems on Bitcoin, [79, 87] allow users to log statements on the blockchain; [36] is a key-value database with get/set operations; [50] extends Bitcoin with advanced financial operations (like e.g., creation of virtual assets, payment of dividends, *etc.*), by embedding its own messages in Bitcoin transactions.

Unfortunately, since the Bitcoin language is not Turing-complete, the possible contracts that are natively supported the platform are very limited. However, the Bitcoin protocol allows clients to embed a few extra bytes as metadata in transactions. Many platforms for contracts exploit these metadata to store a persistent, timestamped and tamper-proof historical record of all their messages [1, 24]. Usually, metadata are stored in `OP_RETURN` transactions [25, 2], making them meaningless to the Bitcoin network. With this approach, the sequence of platform-dependent messages forms a *subchain*, whose content can only be interpreted by the nodes that execute the platform, thus potentially extending the decentralization power of Bitcoin to systems that use more complex contract models.

As far as we know, none of the existing platforms use a secure protocol to establish if their subchain is consistent. This is a serious issue, because it either limits the expressiveness of the contracts supported by these platforms (which must consider all messages as consistent, so basically losing the notion of state), or degrades the security of contracts (because adversaries can manage to publish inconsistent messages, so tampering with the execution of smart contracts).

1.2 Contributions

Pursuing the ideas described in the previous section, this thesis presents two main scientific contributions: first, a centralized contract broker in the form of a contract-oriented middleware, and, second, a protocol for extending the contract models supported by Bitcoin through subchains, providing similar security properties to those given by the native Bitcoin consensus mechanism.

1.2.1 A contract-oriented middleware

In Chapter 3 we present a design, an implementation[1], and a validation of a centralized middleware which uses behavioural contracts to allow disciplined interactions between mutually distrusting services. In particular, the middleware is designed to support different notions of contract, which only need to share some high-level features:

- a *compliance* relation between contracts, which specifies when services conforming to their contracts interact correctly. The middleware guarantees that only services with compliant contracts can interact.
- an *execution monitor*, which checks if the actions done by the services conform to their contracts, and — otherwise — detects which services are *culpable* of a contract violation.

Building upon these basic ingredients, the middleware extends standard message-oriented middleware [15] (MOMs) by allowing services to advertise contracts, establish sessions between services with compliant contracts, and interact through these sessions. The execution monitor guarantees that, whenever a contract is violated, the culprit is sanctioned. Sanctions negatively affect the reputation of a service, and consequently its chances to establish new sessions.

We also explore several ways to validate our middleware. First, we perform some scalability tests, to measure the execution time of the core primitives as a function of the number of advertised contracts. Second, we develop a distributed application (to solve an RSA factoring challenge [77]), involving a master and

[1] Available at `co2.unica.it`.

a population of workers, some of which do not always respect their contracts. In particular, we show that our service selection mechanism allows to automatically marginalize the dishonest services, without requiring the master to explicitly handle their misbehaviour. In the last part of this contribution, we introduce a hands-on approach to contract-oriented computing, showing some programming patterns and code snippets that exploits the features of our middleware to write a (contract-based) distributed application.

1.2.2 A protocol for contracts on Bitcoin

In Chapter 4, we present a protocol that allows third-party applications to keep trace of their contract executions on the Bitcoin blockchain, overcoming the limit of trusting in a single entity (as a centralized middleware). To this purpose, we first formalise the concept of subchain, and then present a notion of subchain consistency. We show that subchains are suitable to embed state updates of a general *labelled transition system* (LTS), as a basis for implementing behavioural contracts upon the Bitcoin blockchain.

We provide a wide description of the protocol, presenting its basic properties. We say that the protocol is based on a *Proof-of-Stake* [33, 63] consensus, similar to the Proof-of-Work used by Bitcoin, but where nodes votes are weighted by the money stake they own, instead of their computational power. Moreover, our protocol is organized in stages: at each stage, a new message to be appended to the subchain is randomly chosen. Intuitively, a node must endorse (or *vote*) a message, in order to candidate it to be appendend on the subchain in the current stage. To vote, it simply puts a bitcoin deposit on it. Then, if the message is chosen, the rest of the network can verify its consistency, paying back the associated deposit to its voter. Since clients pay a fee to the nodes that endorse their messages, and since the deposit and the fee are scrutinized by the network, the protocol provides an economic incentive to honest nodes that vote consistent updates only, while disincentivizing the dishonest ones.

We analytically and experimentally validate the security of our protocol, in presence of colluding adversaries that try to maximize their revenue generated from the participation in the protocol, and that try to publish both consistent and in-

consistent messages, depending on the convenience. In particular, we show that under conservative assumptions, the protocol places a lower bound to the stake that adversaries should globally own in order to subvert the system of incentives.

Finally, we provide a description of a concrete implementation of the protocol in Bitcoin. Notably, our protocol can be implemented by only using the so-called *standard* transactions[2]. A preliminary implementation prototype of our protocol is public available[3].

1.3 Synopsis

This dissertation presents both unpublished material, and some published one. We briefly describe below its organization.

Chapter 2. *Background* introduces some of the theory, basic notions and part of the notations used throughout our subsequent technical development. In particular, the theory of *timed automata* and *timed session types*, used in the subsequent Chapter 3, is presented; it also illustrates *timed* CO_2, a calculus for contract-oriented application. Finally, it provides an overview of Bitcoin and its blockchain.

Chapter 3. *A contract-oriented middleware* presents the first main contribution of this thesis: a centralized Java middleware that allows mutually distrusting services to safely interact through contracts. In particular, the last sections of this chapter show the middleware features and the good programming patterns to develop contract-oriented applications. This chapter borrows some material from [18, 19, 10].

Chapter 4. *Decentralizing behavioural contracts on Bitcoin* presents the second main contribution of this work: a protocol that allows third-party platforms to safely execute behavioural contracts on decentralized contexts, thus overcoming the

[2]This is important, because non-standard transactions are discarded by nodes running the official Bitcoin client.

[3]At http://contractvm.github.io/.

need for a centralized trusted middleware. To do so, the protocol extends the consensus mechanism of Bitcoin with the idea of subchains, to support non-native advanced models of contracts. This chapter widely extends the material in [22].

Chapter 5. *Related work* illustrates several state-of-the-art studies related to our thesis. More precisely, it discusses different models and architectures for contract-oriented applications, both in the general context and in decentralized environments like Bitcoin. It also presents an overview of Ethereum, a cryptocurrency similar to Bitcoin, which natively supports an expressive model of contracts.

Chapter 6. *Conclusions* finally contains a summarized view of our work, a discussion of the results of the two main contributions, and some research proposals for further work.

Chapter 2

Background

In this chapter, we illustrate some theory and notions that will be used in the prosecution of the thesis. In particular, next Section 2.2 illustrates the theory of timed automata, that the middleware exploits to monitor, at run-time, the interaction between compliant services. Then, in Section 2.1 and Section 2.3 we introduce timed session types, a notable contract model that we use in the presentation of the middleware, and TCO_2 a calculus that allow to specify contract-oriented applications. In Section 2.4, we briefly describe Bitcoin, the blockchain and its consensus mechanism: this concepts will be useful to understand the core features of our protocol for executing behavioural contracts in decentralized environments.

Except for Section 2.4, this chapter does not provide original contributions from the author.

2.1 Timed Session Types (TSTs)

Although the middleware described in Chapter 3 is meant to be contract-agnostic, we use *timed session types* (TSTs) as the referring contract model used for its presentation and validation. Therefore, in the following sections we illustrate the core notions of this model. Let $\mathtt{A}$ be a set of *actions*, ranged over by $\mathtt{a}, \mathtt{b}, \ldots$. We denote with $\mathtt{A}^!$ the set $\{!\mathtt{a} \mid \mathtt{a} \in \mathtt{A}\}$ of *output actions*, with $\mathtt{A}^?$ the set $\{?\mathtt{a} \mid \mathtt{a} \in \mathtt{A}\}$ of *input actions*, and with $\mathtt{L} = \mathtt{A}^! \cup \mathtt{A}^?$ the set of *branch labels*, ranged over by $\ell, \ell', \ldots$. We use $\delta, \delta', \ldots$ to range over the set $\mathbb{R}_{\geq 0}$ of positive real numbers including zero,

and $d, d', \ldots$ to range over the set $\mathbb{N}$ of natural numbers. Let $\mathbb{C}$ be a set of *clocks*, namely variables in $\mathbb{R}_{\geq 0}$, ranged over by $t, t', \ldots$. We use $R, T, \ldots \subseteq \mathbb{C}$ to range over sets of clocks.

Definition 2.1.1 (Guard)**.** We define the set $\mathcal{G}_{\mathbb{C}}$ of *guards* over clocks $\mathbb{C}$ as follows:

$$g \;::=\; \mathtt{true} \;\mid\; \neg g \;\mid\; g \wedge g \;\mid\; t \circ d \;\mid\; t - t' \circ d \qquad \text{where } \circ \in \{<, \leq, =, \geq, >\}$$

A TST p models the behaviour of a single participant involved in an interaction (Definition 2.1.2). To give some intuition, we consider two participants, Alice (A) and Bob (B), who want to interact. A uses an *internal choice* in the form $\sum_i \mathtt{!a}_i\{g_i, R_i\}\,.\,p_i$ when she wants to do one of the outputs $\mathtt{!a}_i$ in a time window where g_i is true; further, the clocks in R_i will be reset after the output is performed. Dually, B uses an *external choice* in the form $\&_i \mathtt{?a}_i\{g_i, R_i\}\,.\,q_i$ to declare that he is available to receive each message $\mathtt{a}_i$ in *any instant* within the time window defined by g_i (and the clocks in R_i will be reset after the input). In both cases, the contract execution continues with p_i.

Definition 2.1.2 (Timed session type)**.** *Timed session types* $p, q, \ldots$ are terms of the following grammar:

$$p \;::=\; \mathbf{1} \;\mid\; \textstyle\sum_{i \in I} \mathtt{!a}_i\{g_i, R_i\}\,.\,p_i \;\mid\; \&_{i \in I} \mathtt{?a}_i\{g_i, R_i\}\,.\,p_i \;\mid\; \mathrm{rec}\, X\,.\,p \;\mid\; X$$

where (i) the set I is finite and non-empty, (ii) the actions in internal/external choices are pairwise distinct, (iii) recursion is guarded (e.g., we forbid both $\mathrm{rec}\, X\,.\,X$ and $\mathrm{rec}\, X\,.\,\mathrm{rec}\, Y\,.\,p$).

Except where stated otherwise, we consider TSTs up-to unfolding of recursion. A TST is *closed* when it has no recursion variables. If $q = \sum_{i \in I} \mathtt{!a}_i\{g_i, R_i\}\,.\,p_i$ and $0 \notin I$, we write $\mathtt{!a}_0.p_0 + q$ for $\sum_{i \in I \cup \{0\}} \mathtt{!a}_i\{g_i, R_i\}\,.\,p_i$ (the same for external choices). True guards, empty resets, and trailing occurrences of the *success state* $\mathbf{1}$ can be omitted.

Example 2.1.1 (Simplified PayPal)**.** Along the lines of PayPal User Agreement [3], we specify the protection policy for buyers of a simple on-line payment platform,

called PayNow for the full version). PayNow helps customers in on-line purchasing, providing protection against seller misbehaviours. In case a buyer has not received what he has paid for, he can open a dispute within 180 days from the date the buyer made the payment. After opening of the dispute, the buyer and the seller may try to come to an agreement. If this is not the case, within 20 days, the buyer can escalate the dispute to a claim. However, the buyer must wait at least 7 days from the date of payment to escalate a dispute. Upon not reaching an agreement, if still the buyer does not escalate the dispute to a claim within 20 days, the dispute is considered aborted. During a claim procedure, PayNow will ask the buyer to provide documentation to certify the payment, within 3 days of the date the dispute was escalated to a claim. After that, the payment will be refunded within 7 days.

PayNow's agreement is described by the following TST p:

$$
\begin{aligned}
p &= \texttt{?pay}\{t_{pay}\}.\,(\texttt{?ok}\,\&\,\texttt{?dispute}\{t_{pay} < 180, t_d\}.\,p') && \text{where}\\
p' &= \texttt{?ok}\{t_d < 20\}\ \&\\
&\quad\ \texttt{?claim}\{t_d < 20 \wedge t_{pay} > 7, t_c\}.\texttt{?rcpt}\{t_c < 3, t_c\}.\texttt{!refund}\{t_c < 7\}\ \&\\
&\quad\ \texttt{?abort}
\end{aligned}
$$

2.1.1 Semantics of TSTs

To define the behaviour of TSTs we use *clock valuations*, which associate each clock with its value. The state of the interaction between two TSTs is described by a *configuration* $(p, \nu) \mid (q, \eta)$, where the clock valuations ν and η record (keeping the same pace) the time of the clocks in p and q, respectively. The dynamics of the interaction is formalised as a transition relation between configurations (Definition 2.1.6). This relation describes all and only the *correct* interactions: e.g., we do not allow time passing to make unsatisfiable all the guards in an internal choice, since doing so would prevent a participant from respecting her protocol.

Definition 2.1.3 (Clock valuations)**.** We denote with $\mathbb{V} = \mathbb{C} \to \mathbb{R}_{\geq 0}$ the set of *clock valuations*. We use meta-variables $\nu, \eta, \ldots$ to range over $\mathbb{V}$, and we denote with ν_0, η_0 the *initial* clock valuations, which map each clock to zero. Given a clock

valuation ν, we define the following valuations:

- $\nu + \delta$ increases each clock in ν by the delay $\delta \in \mathbb{R}_{\geq 0}$, i.e.:

$$(\nu + \delta)(t) = \nu(t) + \delta \qquad \text{for all } t \in \mathbb{C}$$

- $\nu[R]$ *resets* all the clocks in the set $R \subseteq \mathbb{C}$, i.e.:

$$\nu[R](t) = \begin{cases} 0 & \text{if } t \in R \\ \nu(t) & \text{otherwise} \end{cases}$$

Definition 2.1.4 (Semantics of guards)**.** For all guards g, we define the set of clock valuations $[\![g]\!]$ inductively as follows, where $\circ \in \{<, \leq, =, \geq, >\}$:

$$[\![\mathtt{true}]\!] = \mathbb{V} \qquad [\![\neg g]\!] = \mathbb{V} \setminus [\![g]\!] \qquad [\![g_1 \wedge g_2]\!] = [\![g_1]\!] \cap [\![g_2]\!]$$
$$[\![t \circ d]\!] = \{\nu \mid \nu(t) \circ d\} \qquad [\![t - t' \circ d]\!] = \{\nu \mid \nu(t) - \nu(t') \circ d\}$$

Before defining the semantics of TSTs, we recall from [32] some basic operations on *sets* of clock valuations (ranged over by $\mathcal{K}, \mathcal{K}', \ldots \subseteq \mathbb{V}$).

Definition 2.1.5 (Past and inverse reset)**.** For all sets $\mathcal{K}$ of clock valuations, the set of clock valuations $\downarrow \mathcal{K}$ (the *past* of $\mathcal{K}$) and $\mathcal{K}[T]^{-1}$ (the *inverse reset* of $\mathcal{K}$) are defined as:

$$\downarrow \mathcal{K} \;=\; \{\nu \mid \exists \delta \geq 0 : \nu + \delta \in \mathcal{K}\} \qquad \mathcal{K}[T]^{-1} \;=\; \{\nu \mid \nu[T] \in \mathcal{K}\}$$

Definition 2.1.6 (Semantics of TSTs)**.** A *configuration* is a term of the form $(p, \nu) \mid (q, \eta)$, where p, q are TSTs extended with *committed choices* $[\,!\mathtt{a}\{g, R\}\,]\,p$. The semantics of TSTs is a labelled relation $\rightarrow$ over configurations (Figure 2.1), whose labels are either silent actions τ, delays δ, or branch labels, and where we define the set of clock valuations $\mathtt{rdy}(p)$ as:

$$\mathtt{rdy}(p) \;=\; \begin{cases} \downarrow \bigcup [\![g_i]\!] & \text{if } p = \sum_{i \in I} !\mathtt{a}_i\{g_i, R_i\} \,.\, p_i \\ \mathbb{V} & \text{if } p = \,\&\cdots \text{ or } p = \mathbf{1} \\ \emptyset & \text{otherwise} \end{cases}$$

$$(\texttt{!a}\{g, R\}.\, p{+}p',\ \nu) \xrightarrow{\tau} ([\texttt{!a}\{g, R\}]\, p,\ \nu) \qquad \text{if } \nu \in [\![g]\!] \qquad [+]$$

$$([\texttt{!a}\{g, R\}]\, p,\ \nu) \xrightarrow{\texttt{!a}} (p,\ \nu[R]) \qquad [!]$$

$$(\texttt{?a}\{g, R\}.\, p + p',\ \nu) \xrightarrow{\texttt{?a}} (p,\ \nu[R]) \qquad \text{if } \nu \in [\![g]\!] \qquad [?]$$

$$(p,\ \nu) \xrightarrow{\delta} (p,\ \nu + \delta) \qquad \text{if } \delta > 0 \ \wedge\ \nu + \delta \in \texttt{rdy}(p) \qquad [\textsc{Del}]$$

$$\frac{(p, \nu) \xrightarrow{\tau} (p', \nu')}{(p, \nu) \mid (q, \eta) \xrightarrow{\tau} (p', \nu') \mid (q, \eta)}\ [\text{S-}+] \qquad \frac{(p, \nu) \xrightarrow{\delta} (p, \nu') \quad (q, \eta) \xrightarrow{\delta} (q, \eta')}{(p, \nu) \mid (q, \eta) \xrightarrow{\delta} (p, \nu') \mid (q, \eta')}\ [\text{S-}\textsc{Del}]$$

$$\frac{(p, \nu) \xrightarrow{\texttt{!a}} (p', \nu') \quad (q, \eta) \xrightarrow{\texttt{?a}} (q', \eta')}{(p, \nu) \mid (q, \eta) \xrightarrow{\tau} (p', \nu') \mid (q', \eta')}\ [\text{S-}\tau]$$

Figure 2.1: Semantics of timed session types (symmetric rules omitted).

As usual, we write $p \xrightarrow{\alpha} p'$ as a shorthand for $(p, \alpha, p') \in \rightarrow$, with $\alpha \in \mathtt{L} \cup \{\tau\} \cup \mathbb{R}_{\geq 0}$. Given a relation $\rightarrow$, we denote with $\rightarrow^*$ its reflexive and transitive closure.

We now comment the rules in Figure 2.1. The first four rules describe the behaviour of a TST in isolation. Rule [+] allows a TST to commit to the branch $\texttt{!a}$ of her internal choice, provided that the corresponding guard is satisfied in the clock valuation ν. This results in the term $[\texttt{!a}\{g, R\}]\, p$, which can only fire $\texttt{!a}$ through rule [!], without making time pass. This term represents a state where the endpoint has committed to branch $\texttt{!a}$ in a specific time instant[1]. Rule [?] allows an external choice to fire any of its input actions whose guard is satisfied. Rule [Del] allows time to pass; this is always possible for external choices and success term, while for an internal choice we require that at least one of the guards remains satisfiable; this is obtained through the function $\texttt{rdy}$. The last three rules deal with configurations. Rule [S-+] allows a TST to commit in an internal choice. Rule [S-τ] is the standard synchronisation rule *à la* CCS. Rule [S-Del] allows time to pass, equally for both endpoints.

Example 2.1.2. Let $p = \texttt{!a}{+}\texttt{!b}\{t > 2\}$, let $q = \texttt{?b}\{t > 5\}$, and consider the

[1] This is quite similar to the handling of internal choices in [16, 20]. In these works, an internal choice $\texttt{!a}.p{+}q$ first commits to one of the branches (say, $\texttt{!a}.p$) through an internal action, taking a transition to a singleton internal choice $\texttt{!a}.p$. In this state, only the action $\texttt{!a}$ is enabled — as in our $[\texttt{!a}\{g, R\}]\, p$.

computations:

$$(p, \nu_0) \mid (q, \eta_0) \xrightarrow{7}\xrightarrow{\tau} ([\texttt{!b}\{t > 2\}], \nu_0 + 7) \mid (q, \eta_0 + 7) \xrightarrow{\tau} (\mathbf{1}, \nu_0 + 7) \mid (\mathbf{1}, \eta_0 + 7) \tag{2.1}$$

$$(p, \nu_0) \mid (q, \eta_0) \xrightarrow{\delta}\xrightarrow{\tau} ([\texttt{!a}], \nu_0 + \delta) \mid (q, \eta_0 + \delta) \tag{2.2}$$

$$(p, \nu_0) \mid (q, \eta_0) \xrightarrow{3}\xrightarrow{\tau} ([\texttt{!b}\{t > 2\}], \nu_0 + 3) \mid (q, \eta_0 + 3) \tag{2.3}$$

The computation in (2.1) reaches success, while the other two computations reach a deadlock state. In (2.2), p commits to the choice $\texttt{!a}$ after some delay δ; at this point, time cannot pass (because the leftmost endpoint is a committed choice), and no synchronisation is possible (because the other endpoint is not offering $\texttt{?a}$). In (2.3), p commits to $\texttt{!b}$ after 3 time units; here, the rightmost endpoint would offer $\texttt{?b}$, but not in the time chosen by the leftmost endpoint. Note that, were we allowing time to pass in committed choices, then we would have obtained e.g. that $(\texttt{!b}\{t > 2\}, \nu_0) \mid (q, \eta_0)$ never reaches deadlock — contradicting our intuition that these endpoints should not be considered compliant.

Note that, even when p and q have shared clocks, the rules in Figure 2.1 ensure that there is no interference between them. For instance, if a transition of (p, ν) resets some clock t, this has no effect on a clock with the same name in q, i.e. on a transition of $(p, \nu) \mid (q, \eta)$. Thus, without loss of generality we will assume that the clocks in p and in q are disjoint.

2.1.2 Compliance between TSTs

We extend to the timed setting the standard progress-based compliance between (untimed) session types [17, 21, 46, 67]. If p is compliant with q, then whenever an interaction between p and q becomes stuck, it means that both participants have reached the success state. Intuitively, when two TSTs are compliant, the interaction between services correctly implementing[2] them will progress (without communication and time errors), until both services reach a success state.

[2]The notion of "correct implementation" of a TST is orthogonal to the present work. A possible instance of this notion can be obtained by extending to the timed setting the *honesty* property of [26].

Definition 2.1.7 (Compliance). We say that $(p, \nu) \mid (q, \eta)$ is *deadlock* whenever (i) it is not the case that both p and q are $\mathbf{1}$, and (ii) there is no δ such that $(p, \nu + \delta) \mid (q, \eta + \delta) \xrightarrow{\tau}$. We then write $(p, \nu) \bowtie (q, \eta)$ whenever:

$$(p, \nu) \mid (q, \eta) \to^* (p', \nu') \mid (q', \eta') \quad \text{implies} \quad (p', \nu') \mid (q', \eta') \text{ not deadlock}$$

We say that p and q are *compliant* whenever $(p, \nu_0) \bowtie (q, \eta_0)$ (in short, $p \bowtie q$).

Note that item (ii) of the definition of deadlock can be equivalently phrased as follows: $(p, \nu) \mid (q, \eta) \not\xrightarrow{\tau}$ (i.e., the configuration cannot do a τ-move in the *current* clock valuation), and there does not exist any $\delta > 0$ such that $(p, \nu) \mid (q, \eta) \xrightarrow{\delta} \xrightarrow{\tau}$.

Example 2.1.3. The TSTs $p = \texttt{?a}\{t < 5\}.\texttt{!b}\{t < 3\}$ and $q = \texttt{!a}\{t < 2\}.\texttt{?b}\{t < 3\}$ are compliant, but p is *not* compliant with $q' = \texttt{!a}\{t < 5\}.\texttt{?b}\{t < 3\}$. Indeed, if q' outputs `a` at, say, time 4, the configuration will reach a state where no actions are possible, and time cannot pass. This is a deadlocked state, according to Definition 2.1.7.

Example 2.1.4. Consider a customer of PayNow (Example 2.1.1) who is willing to wait 10 days to receive the item she has paid for, but after that she will open a claim. Further, she will instantly provide PayNow with any documentation required. The customer contract is described by the following TST, which is compliant with PayNow's p in Example 2.1.1:

$$\texttt{!pay}\{t_{pay}\}.(\ \texttt{!ok}\{t_{pay} < 10\} + \texttt{!dispute}\{t_{pay} = 10\}.\texttt{!claim}\{t_{pay} = 10\}.\texttt{!rcpt}\{t_{pay} = 10\}.\texttt{?refund})$$

Compliance between TSTs is more liberal than the untimed notion, as it can relate terms which, when cleaned from all the time annotations, would not be compliant in the untimed setting.

Definition 2.1.8 and Lemma 2.1.1 below coinductively characterise compliance between TSTs, by extending to the timed setting the coinductive compliance for untimed session types in [16]. Intuitively, an internal choice p is compliant with q when (i) q is an external choice, (ii) for each output `!a` that p can fire after δ time units, there exists a corresponding input `?a` that q can fire after δ time units, and (iii) their continuations are coinductively compliant. The case where p is an

external choice is symmetric.

Definition 2.1.8. We say $\mathcal{R}$ is a *coinductive compliance* iff $(p, \nu)\ \mathcal{R}\ (q, \eta)$ implies:

1. $p = \mathbf{1} \iff q = \mathbf{1}$
2. $p = \sum_{i \in I} !\mathtt{a}_i\{g_i, R_i\}\,.\,p_i \implies \nu \in \mathtt{rdy}(p) \ \wedge\ q = \&_{j \in J} ?\mathtt{a}_j\{g_j, R_j\}\,.\,q_j \ \wedge$
   $\forall \delta, i : \nu + \delta \in [\![g_i]\!] \implies \exists j : \mathtt{a}_i = \mathtt{a}_j \wedge \eta + \delta \in [\![g_j]\!] \wedge (p_i, \nu + \delta[R_i])\mathcal{R}(q_j, \eta + \delta[R_j])$
3. $p = \&_{j \in J} ?\mathtt{a}_j\{g_j, R_j\}\,.\,p_j \implies \eta \in \mathtt{rdy}(q) \ \wedge\ q = \sum_{i \in I} !\mathtt{a}_i\{g_i, R_i\}\,.\,q_i \ \wedge$
   $\forall \delta, i : \eta + \delta \in [\![g_i]\!] \implies \exists j : \mathtt{a}_i = \mathtt{a}_j \wedge \nu + \delta \in [\![g_j]\!] \wedge (p_j, \nu + \delta[R_j])\mathcal{R}(q_i, \eta + \delta[R_i])$

Lemma 2.1.1. $p \bowtie q \iff \exists\, \mathcal{R}$ *coinductive compliance* $:\ (p, \nu_0)\ \mathcal{R}\ (q, \eta_0)$

Theorem 2.1.1. *Compliance between TSTs is decidable.* [26]

2.1.3 Admissibility of a compliant

In the untimed setting, each session type p admits a compliant, i.e. there exists some q such that $p \bowtie q$. For instance, we can compute q by simply swapping internal choices with external ones (and inputs with outputs) in p (this q is called the *canonical dual* of p in some papers [45, 54]). A naïve attempt to extend this construction to TSTs can be to swap internal with external choices, as in the untimed case, and leave guards and resets unchanged. This construction does not work as expected, as shown by the following example.

Example 2.1.5. Consider the following TSTs:

$$p_1 = !\mathtt{a}\{x \leq 2\}.\,!\mathtt{b}\{x \leq 1\} \qquad p_2 = !\mathtt{a}\{x \leq 2\} + !\mathtt{b}\{x \leq 1\}.\,?\mathtt{a}\{x \leq 0\}$$
$$p_3 = \mathrm{rec}\, X\,.\,?\mathtt{a}\{x \leq 1 \wedge y \leq 1\}.\,!\mathtt{a}\{x \leq 1, \{x\}\}.\,X$$

The TST p_1 is not compliant with its naïve dual $q_1 = ?\mathtt{a}\{x \leq 2\}.\,?\mathtt{b}\{x \leq 1\}$: even though q_1 can do the input $?\mathtt{a}$ in the required time window, p_1 cannot perform $!\mathtt{b}$ if $!\mathtt{a}$ is performed after 1 time unit. For this very reason, no TST is compliant with p_1. Note instead that $q_1 \bowtie !\mathtt{a}\{x \leq 1\}.\,!\mathtt{b}\{x \leq 1\}$, which is *not* its naïve dual. In p_2, a similar deadlock situation occurs if the $!\mathtt{b}$ branch is chosen, and so also p_2 does not admit a compliant. The reason why p_3 does not admit a compliant is

$$\Gamma \vdash \mathbf{1} : \mathbb{V} \qquad \text{[T-1]}$$

$$\frac{\Gamma \vdash p_i : \mathcal{K}_i \qquad \forall i \in I}{\Gamma \vdash \&_{i\in I} ?\mathtt{a}_i\{g_i, T_i\}\,.\,p_i : \bigcup_{i\in I} {\downarrow}\left([\![g_i]\!] \cap \mathcal{K}_i[T_i]^{-1}\right)} \qquad \text{[T-\&]}$$

$$\frac{\Gamma \vdash p_i : \mathcal{K}_i \qquad \forall i \in I}{\Gamma \vdash \sum_{i\in I} !\mathtt{a}_i\{g_i, T_i\}\,.\,p_i : \left(\bigcup_{i\in I} {\downarrow}[\![g_i]\!]\right) \setminus \left(\bigcup_{i\in I} {\downarrow}\left([\![g_i]\!] \setminus \mathcal{K}_i[T_i]^{-1}\right)\right)} \qquad \text{[T-+]}$$

$$\Gamma, X : \mathcal{K} \vdash X : \mathcal{K} \qquad \text{[T-Var]}$$

$$\frac{\Gamma, X : \mathcal{K} \vdash p : \mathcal{K}'}{\Gamma \vdash \mathrm{rec}\, X\,.\,p : \bigcup\{\mathcal{K}_0 \mid \exists \mathcal{K}_1 :\ \Gamma, X : \mathcal{K}_0 \vdash p : \mathcal{K}_1 \wedge \mathcal{K}_0 \subseteq \mathcal{K}_1\}} \qquad \text{[T-Rec]}$$

Figure 2.2: Kind system for TSTs.

more subtle: actually, p_3 can loop until the clock y reaches the value 1; after this point, the guard $y \leq 1$ can no longer be satisfied, and then p_3 reaches a deadlock.

To establish when a TST admits a compliant, we define a kind system which associates to each p a set of clock valuations $\mathcal{K}$ (called *kind of* p). The kind of a TST is unique, and each closed TST is kindable (Theorem 2.1.2). If p has kind $\mathcal{K}$, then there exists some q such that, for all $\nu \in \mathcal{K}$, the configuration $(p, \nu) \mid (q, \nu)$ never reaches a deadlock (Theorem 2.1.3). Also the converse statement holds: if, for some q, $(p, \nu) \mid (q, \nu)$ never reaches a deadlock, then $\nu \in \mathcal{K}$ (Theorem 2.1.4). Therefore, p admits a compliant whenever the initial clock valuation ν_0 belongs to $\mathcal{K}$. We give a constructive proof of the correctness of the kind system, by showing a TST $\mathrm{co}(p)$ which we call the *canonical compliant* of p.

Definition 2.1.9 (Kind system for TSTs)**.** Kind judgements $\Gamma \vdash p : \mathcal{K}$ are defined in Figure 2.2, where Γ is a partial function which associates kinds to recursion variables.

Rule [T-1] says that the success TST $\mathbf{1}$ admits a compliant in every ν: indeed, $\mathbf{1}$ is compliant with itself. The kind of an external choice is the union of the kinds of its branches (rule [T-&]), where the kind of a branch is the past of those clock valuations which satisfy both the guard and, after the reset, the kind of their continuation. Internal choices are dealt with by rule [T-+], which computes the difference between the union of the past of the guards and a set of error clock valuations. The error clock valuations are those which can satisfy a guard but not

the kind of its continuation. Rule [T-VAR] is standard. Rule [T-REC] looks for a kind which is preserved by unfolding of recursion (hence a fixed point).

The following theorem states that *every* closed TST is kindable, as well as uniqueness of kinding. We stress that being kindable does not imply admitting a compliant: this holds if and only if the initial clock valuation ν_0 belongs to the kind. Note that uniqueness of kinding holds at the *semantic* level, but the same kind can be represented syntactically in different ways.

Theorem 2.1.2 (Uniqueness of kinding)**.** *For all* p *and* Γ *with* $\mathtt{fv}(p) \subseteq \mathrm{dom}(\Gamma)$*, there exists unique* $\mathcal{K}$ *such that* $\Gamma \vdash p : \mathcal{K}$*.*

By exploiting the kind system we define the *canonical compliant* of kindable TSTs. Roughly, we turn internal choices into external ones (without changing guards nor resets), and external into internal, changing the guards so that the kind of continuations is preserved.

Example 2.1.6 (Canonical compliant)**.** For all kinding environments Γ and p kindable in Γ, we define the TST $\mathrm{co}_\Gamma(p)$. We will abbreviate $\mathrm{co}_\Gamma(p)$ as $\mathrm{co}(p)$ when $\Gamma = \emptyset$.

The following theorem states the soundness of the kind system: is particular, if the initial clock valuation ν_0 belongs to the kind of p, then p admits a compliant.

Theorem 2.1.3 (Soundness)**.** *If* $\vdash p : \mathcal{K}$ *and* $\nu \in \mathcal{K}$*, then* $(p, \nu) \bowtie (\mathrm{co}(p), \nu)$*.*

Example 2.1.7. Recall $q_1 = \mathtt{?a}\{x \leq 2\}.\,\mathtt{?b}\{x \leq 1\}$ from Example 2.1.5. We have $\mathrm{co}(q_1) = \mathtt{!a}\{x \leq 1\}.\,\mathtt{!b}\{x \leq 1\}$. Since $\vdash q_1 : \mathcal{K} = [\![x \leq 1]\!]$ and $\nu_0 \in \mathcal{K}$, by Theorem 2.1.3 we have that $q_1 \bowtie \mathrm{co}(q_1)$, as anticipated in Example 2.1.5.

The following theorem states the kind system is also *complete*: in particular, if p admits a compliant, then the clock valuation ν_0 belongs to the kind of p.

Theorem 2.1.4 (Completeness)**.** *If* $\vdash p : \mathcal{K}$ *and* $\exists q, \eta.\ (p, \nu) \bowtie (q, \eta)$*, then* $\nu \in \mathcal{K}$*.*

Compliance is not transitive, in general: however, Theorem 2.1.5 below states that transitivity holds when passing through the canonical compliant.

Lemma 2.1.2. *For all* p, q, ν, η *and* p', ν' *such that* $\vdash p' : \mathcal{K}$ *and* $\nu' \in \mathcal{K}$*:*

$$(p, \nu) \bowtie (p', \nu') \ \wedge \ (\mathrm{co}(p'), \nu') \bowtie (q, \eta) \implies (p, \nu) \bowtie (q, \eta)$$

Theorem 2.1.5 (Transitivity of compliance)**.** *If* $p \bowtie p'$ *and* $\mathrm{co}(p') \bowtie q$*, then* $p \bowtie q$*.*

2.1.4 Runtime monitoring

We now define the semantics of the runtime monitor of TSTs, which is the one used in the premises of rules [SEND] and [RECV] in Figure 2.7. Note that the semantics in Figure 2.1 cannot be directly exploited to define such a runtime monitor, for two reasons. First, the synchronisation rule is symmetric and synchronous, while the middleware assumes an asymmetry between internal and external choices and an asynchronous semantics. Second, the semantics in Figure 2.1 does not have transitions (either messages or delays) not allowed by the TSTs, while the monitoring semantics must also consider illegal moves attempted by participants.

The monitoring semantics is defined on two levels. The first level, specified by the relation $\rightarrow$ (which overloads the transition relation used in Section 2.1.1) deals with the case of honest participants; however, unlike the semantics in Section 2.1.1, here we decouple the action of sending from that of receiving. More precisely, if A has an internal choice and B has an external choice, then we postulate that A must move first, by doing one of the outputs in her choice, and then B must be ready to do the corresponding input. The second level, called *monitoring semantics* and specified by the relation $\twoheadrightarrow$, builds upon the first one to allow for synchronisation and delay. Additionally, the monitoring semantics defines transitions for actions not accepted by the first level, e.g. unexpected input/output actions. In these cases, the monitoring semantics assigns the blame to the culpable participant, by setting its state to $\mathbf{0}$.

Definition 2.1.10 (Monitoring semantics of TSTs)**.** *Monitoring configurations* $\gamma, \gamma', \ldots$ are terms of the form $P \parallel Q$, P and Q are triples (p, c, ν), where p is either a TST or $\mathbf{0}$, and c is a sequence of output labels (possibly empty). The transition relations $\rightarrow$ and $\twoheadrightarrow$ over monitoring configurations, with labels $\lambda, \lambda', \ldots \in (\{\mathsf{A}, \mathsf{B}\} \times \mathtt{L}) \cup \mathbb{R}_{\geq 0}$, is defined in Figure 2.3.

$$(\texttt{!a}\{g, R\}.\, p + p', c, \nu) \xrightarrow{\texttt{!a}} (p, c \cdot \texttt{!a}, \nu[R]) \qquad \text{if } \nu \in [\![g]\!] \qquad \text{[M-+]}$$

$$(\texttt{?a}\{g, R\}.\, p \;\&\; p', c, \nu) \xrightarrow{\texttt{?a}} (p, c, \nu[R]) \qquad \text{if } \nu \in [\![g]\!] \qquad \text{[M-\&]}$$

$$\frac{\nu + \delta \in \texttt{rdy}(p)}{(p, c, \nu) \xrightarrow{\delta} (p, c, \nu + \delta)} \qquad \text{[M-Del]}$$

$$\frac{\nu + \delta \notin \texttt{rdy}(p)}{(p, c, \nu) \xrightarrow{\delta} (\mathbf{0}, c, \nu + \delta)} \qquad \text{[M-DelFail]}$$

$$\frac{(p, c, \nu) \xrightarrow{\texttt{!a}} (p', c', \nu') \qquad (q, d, \eta) \xrightarrow{\texttt{?a}} (q', d', \eta')}{(p, c, \nu) \parallel (q, d, \eta) \xrightarrow{\texttt{A:!a}} (p', c', \nu') \parallel (q', d', \eta')} \qquad \text{[M-Sync]}$$

$$\frac{(p, c, \nu) \xrightarrow{\delta} (p', c', \nu') \qquad (q, d, \eta) \xrightarrow{\delta} (q', d', \eta')}{(p, c, \nu) \parallel (q, d, \eta) \xrightarrow{\delta} (p', c', \nu') \parallel (q', d', \eta')} \qquad \text{[M-SyncDel]}$$

$$(p, \texttt{!a} \cdot c, \nu) \parallel (q, d, \eta) \xrightarrow{\texttt{B:?a}} (p, c, \nu) \parallel (q, d, \eta) \qquad \text{[M-Read]}$$

$$\frac{(p, c, \nu) \overset{\texttt{!a}}{\not\longrightarrow}}{(p, c, \nu) \parallel (q, d, \eta) \xrightarrow{\texttt{A:!a}} (\mathbf{0}, c, \nu) \parallel (q, d, \eta)} \qquad \text{[M-Fail]}$$

Figure 2.3: Monitoring semantics (symmetric rules omitted).

In the rules in Figure 2.3, we always assume that the leftmost TST is governed by A, while the rightmost one is governed by B. In rule [M-+], A has an internal choice, and she can fire one of her outputs !a, provided that the guard g is satisfied. When this happens, the message !a is written to the buffer, and the clocks in R are reset. In rule [M-&], B can enable an input ?a in an external choice; this is permitted when the guard g of the selected branch is satisfied. Rules [M-Del] and [M-DelFail] allow time to pass, making A culpable when she definitively disables all the branches in an internal choice. The last four rules specify the runtime monitor. Rule [M-Sync] allows two triples to synchronise; this makes the buffer of A grow (!a is enqueued, according to rule [M-+]), while B just consumes the input prefix ?a. Rule [M-SyncDel] lets some time δ to pass, provided that the delay is the same for both triples. Rule [M-Read] allows B to read a message in the buffer; note that the state of the recipient is not updated, since the input prefix was already consumed by rule [M-Sync]. Finally, rule [M-Fail] is used when A attempts to do an action not permitted by $\rightarrow$: this makes the monitor evolve to a configuration

where A is culpable (denoted by the term $\mathbf{0}$).

Formally, the runtime monitor can be seen as a *deterministic* automaton, which reads a *timed trace* (a sequence of actions and time delays) and it reaches a unique state γ, which can be inspected to find which of the two participants (if any) is culpable.

Definition 2.1.11 (Duties & culpability)**.** Let $\gamma = (p, c, \nu) \parallel (q, d, \eta)$. We say that A is *culpable* in γ iff $p = \mathbf{0}$. We say that A is *on duty* in γ if (i) A is not culpable in γ, and (ii) either p is an internal choice, or d is not empty.

When both participants behave honestly, i.e., they never take [*Fail*] moves, the monitoring semantics preserves compliance. This can be proved similarly to Theorem 9 in [18].

Example 2.1.8. Let $p = \texttt{!a}\{2 < t < 4\}$ be the TST of participant A, and let $q = \texttt{?a}\{2 < t < 5\} + \texttt{?b}\{2 < t < 5\}$ be that of B. We have that $p \bowtie q$. Let $\gamma_0 = (p, [], \nu_0) \parallel (q, [], \nu_0)$. A correct interaction is given by the timed trace $\eta = \langle 1.2,\ \mathsf{A} : \texttt{!a},\ \mathsf{B} : \texttt{?a} \rangle$. Indeed, $\gamma_0 \xrightarrow{\eta}\!\!\!\!\rightarrow (\mathbf{1}, [], \nu_0) \parallel (\mathbf{1}, [], \nu_0)$. On the contrary, things may go wrong in the following two cases:

(i) a participant does something not permitted. E.g., if A fires a at 1 t.u., by [M-FailA]: $\gamma_0 \xrightarrow{1}\!\!\!\!\rightarrow \xrightarrow{\mathsf{A}:\texttt{!a}}\!\!\!\!\rightarrow (\mathbf{0}, [], \nu_0 + 1) \parallel (q, [], \eta_0 + 1)$, where A is culpable.

(ii) a participant avoids to do something she is supposed to do. E.g., assume that after 6 t.u., A has not yet fired a. By rule [M-SyncDel], we obtain $\gamma_0 \xrightarrow{6}\!\!\!\!\rightarrow (\mathbf{0}, [], \nu_0 + 6) \parallel (q, [], \eta_0 + 6)$, where A is culpable.

2.1.5 Encoding TSTs into Timed Automata

To devise an effective procedure to decide compliance, it is possible to encode TSTs into timed automata, a well-known and more expressive formalism for real-time systems, briefly illustrated in next section.

The mapping from TSTs to TA is omitted in the background of this dissertation, since not relevant for the presentation. We refer to [18, 32, 89, 4, 13] for more details about this topic.

2.2 Timed automata

Timed automata are a classical formalism for modelling real-time systems, introduced by Alur and Dill in the early 90's [5], and extended in many subsequent works [89]. Since then, timed automata have become widely used both in the industry and in the academia, also thanks to successful tools (most notably Uppaal [31]), which enable the modelling and verification of realistic timed systems.

Roughly, a timed automaton (TA) is a finite automaton, annotated with timing contraints and reset predicates using real valued *clocks*.

2.2.1 Basic definitions

Before formalising TA, we introduce some auxiliary notions and notation. Let $\mathbb{C}$ be a set of clocks, ranged over by $t, t', \ldots$ and let $d, d', \ldots$ range over the set $\mathbb{N}$ of natural numbers. We use $R, T, \ldots \subseteq \mathbb{C}$ to range over sets of clocks. Let $\mathtt{A}$ be a set of action names.

We define the set of output action $\mathtt{A}^! = \{\mathtt{!a} \mid \mathtt{a} \in \mathtt{A}\}$ and the set of input actions $\mathtt{A}^? = \{\mathtt{?a} \mid \mathtt{a} \in \mathtt{A}\}$. We denote with $L = \mathtt{A}^! \cup \mathtt{A}^? \cup \{\tau\}$ the set of labels, ranged over by $\ell_\tau, \ell_\tau', \ldots$.

Definition 2.2.1 (Guard)**.** We define the set $\mathcal{G}_\mathbb{C}$ of *guards* over clocks $\mathbb{C}$ as follows:

$$g \ ::= \ \mathtt{true} \ \mid \ \neg g \ \mid \ g \wedge g \ \mid \ t \circ d \ \mid \ t - t' \circ d \qquad \text{where } \circ \in \{<, \leq, =, \geq, >\}$$

The semantics of guards is defined in terms of *clock valuations*: they are functions which associate each clock in $\mathbb{C}$ with a value in $\mathbb{R}_{\geq 0}$. These values are *not* associated with a particular unit of time (i.e., seconds, hours, ...); so, we will call clocks values generically *time units* (abbreviated *t.u.*).

Definition 2.2.2 (Clock valuation)**.** We denote with $\mathbb{V}[\mathbb{C}] : \mathbb{C} \to \mathbb{R}_{\geq 0}$ the set of *clock valuations* over $\mathbb{C}$. We write $\mathbb{V}$ as a shortcut for $\mathbb{V}[\mathbb{C}]$ when $\mathbb{C}$ is clear from the context. We use $\nu, \eta, \ldots$ to range over $\mathbb{V}$, and $\nu_0, \eta_0, \ldots$ to denote the valuations mapping each clock to 0. We use $\mathcal{K}, \mathcal{K}', \ldots$ to range over subsets of $\mathbb{V}$.

Definition 2.2.3 (Time increment and reset)**.** We write $\nu + \delta$ for the clock valuation which increases ν by δ, i.e., for all $t \in \mathbb{C}$:

$$(\nu + \delta)(t) \;=\; \nu(t) + \delta$$

For a set $R \subseteq \mathbb{C}$, we write $\nu[R]$ for the *reset* of the clocks in R, i.e., for all $t \in \mathbb{C}$:

$$\nu[R](t) \;=\; \begin{cases} 0 & \text{if } t \in R \\ \nu(t) & \text{otherwise} \end{cases}$$

When R is a singleton, e.g. $R = \{x\}$, we shall usually write $\nu[x]$ instead of $\nu[\{x\}]$.

Definition 2.2.4. A set of clock valuations $\mathcal{K}$ is said *past closed* if and only if:

$$\nu + \delta \in \mathcal{K} \implies \nu \in \mathcal{K}$$

Definition 2.2.5 (Semantics of guards). For all guards g, we define the set of clock valuations $[\![g]\!]$ inductively as follows, where $\circ \in \{<, \leq, =, \geq, >\}$:

$$[\![\texttt{true}]\!] = \mathbb{V} \qquad [\![\neg g]\!] = \mathbb{V} \setminus [\![g]\!] \qquad [\![g_1 \wedge g_2]\!] = [\![g_1]\!] \cap [\![g_2]\!]$$

$$[\![t \circ d]\!] = \{\nu \mid \nu(t) \circ d\} \qquad [\![t - t' \circ d]\!] = \{\nu \mid \nu(t) - \nu(t') \circ d\}$$

A guard g is said past closed when $[\![g]\!]$ is past closed.

We are now ready to give the definition of timed automata. They are composed by a finite set of *locations*, one of which is the *initial* location. Every location is associated with a past closed guard, called *invariant*, which specifies when the control can be in that location. A subset of locations, the *urgent* locations, are used to model states that "do not let time pass", i.e. force the next transition to be a discrete action. Urgent locations do not add expressive power to the model, since urgency can be specified through invariants [31], and often do not appear in definitions of TA; however, we prefer to have them explicitly.

Locations are connected by *edges*. Basically, edges have a similar role to transition functions of non-deterministic finite states automata, further enriched with a guard specifing when that transition is enabled, and with a set of clocks which

are reset (exactly) when the transition is taken.

Definition 2.2.6 (Timed automaton). A TA is a tuple $A = (Loc, Loc^u, l_0, E, I)$ where: Loc is a finite set of *locations*; $Loc^u \subset Loc$ is the set of *urgent* locations; $l_0 \in Loc$ is the *initial* location; $E \subseteq Loc \times L \times \mathcal{G}_{\mathbb{C}} \times \wp(\mathbb{C}) \times Loc$ is a set of edges; and $I : Loc \to \mathcal{G}_{\mathbb{C}}$ is the *invariant* function. We require that, for all locations $l \in Loc$, $I(l)$ is past closed.

The semantics of TA is defined in terms of *timed* LTSs. These are LTSs much alike those in Section 4.1, except for the fact that, besides actions, labels also include time delays.

Definition 2.2.7 (Timed LTS)**.** A timed labelled transition system (TLTS) is a triple $(Q, \mathrm{L}_\delta, \to)$, where:

- Q is a set (called the set of *configurations*),
- $\mathrm{L}_\delta \supseteq \mathbb{R}_{\geq 0}$ is a set (called set of *labels*, and ranged over by $\alpha, \beta, \ldots$),
- $\to \subseteq Q \times \mathrm{L}_\delta \times Q$ is a relation (called *transition relation*).

An *initial TLTS* is a tuple $(Q, \mathrm{L}_\delta, \to, q_0)$, where $(Q, \mathrm{L}_\delta, \to)$ is a TLTS, and $q_0 \in Q$ is the *initial configuration*.

We are now ready to define the semantics of TA. For the moment, we will assume a TA which runs *in isolation*, i.e. without interacting with other TA.

Definition 2.2.8 (Semantics of timed automata)**.** Let $A = (Loc, Loc^u, l_0, E, I)$ be a TA over a set of clocks $\mathbb{C}$. We define the initial TLTS $[\![A]\!]$ as follows:

$$[\![A]\!] \quad = \quad (Loc \times \mathbb{V}[\mathbb{C}],\ \{\tau\} \cup \mathbb{R}_{\geq 0},\ \to,\ (l_0, \nu_0))$$

where the transition relation $\to$ is specified by the following two rules:

1. $(l, \nu) \xrightarrow{\ell_\tau} (l', \nu')$ if $(l, \ell_\tau, g, R, l') \in E \wedge \nu \in [\![g]\!] \wedge \nu' = \nu[R] \in [\![I(l')]\!]$
2. $(l, \nu) \xrightarrow{\delta} (l, \nu + \delta)$ if $\nu + \delta \in [\![I(l)]\!] \wedge l \notin Loc^u$

A configuration (l, ν) is *reachable* when $(l_0, \nu_0) \to^* (l, \nu)$.

We now comment the two rules in Definition 2.2.8:

- rule 1 allows to perform an action. This does not involve any time delay, but after the action has been performed, all the clocks in R are reset to zero. The action is permitted if the guard g on the edge is satisfied by the current clock valuation, and the invariant $I(l')$ of the target location is satisfied *after* the clock reset.
- rule 2 allows time to pass, provided that this does not break the invariant of the current (not urgent) location. Note that all the clocks progress with the same pace.

2.2.2 Networks of timed automata

We now introduce *networks* of TA, i.e. sets of TA which can interact by synchronizing on channels via input/output actions.

Definition 2.2.9 (**Networks of TA**). A *network of TA* is a finite set of TA (over a given set of clocks $\mathbb{C}$). We denote with $A_1 \mid \cdots \mid A_n$ the network composed by $A_1, \ldots, A_n$.

We now define the semantics of networks of TA. The configurations of a network $A_1 \mid \cdots \mid A_n$ are tuples of the form $(l_1, \ldots, l_n, \nu)$, where $l_1, \ldots, l_n$ represent the current locations in the automata, and ν is an evaluation of *all* the clocks in the network. Similarly to the semantics of isolated TA(Definition 2.2.8), the semantics of a network is a TLTS, whose states are configurations, and labels are internal actions, delays, and channels (for synchronisations).

Definition 2.2.10 (Semantics of networks of TA). Let $A_i = (Loc_i, Loc_i^u, l_0^i, E_i, I_i)$ be a TA over a set of clocks $\mathbb{C}$, for $i \in 1..n$. We define the behaviour of the network $N = A_1 \mid \cdots \mid A_n$ as the initial TLTS $[\![N]\!] = (Q, \mathtt{L}_\delta, \rightarrow, q_0)$, where:

- $Q = Loc_1 \times \cdots \times Loc_n \times \mathbb{V}[\mathbb{C}]$
- $\mathtt{L}_\delta = \mathcal{C} \cup \{\tau\} \cup \mathbb{R}_{\geq 0}$
- $\rightarrow$ is the relation defined in Figure 2.4
- $q_0 = (l_0^1, \ldots, l_0^n, \nu_0)$.

Rule [TAU] states that one of the TA can take an internal edge, provided that its

$$\frac{(l_k, \tau, g, R, l'_k) \in E_k \qquad \nu \in [\![g]\!] \qquad \nu[R] \in [\![\bigwedge_{i\in\{1..n\}\setminus\{k\}} I_i(l_i)]\!] \qquad \nu[R] \in [\![I_k(l'_k)]\!]}{(l_1, \ldots, l_k, \ldots, l_n, \nu) \xrightarrow{\tau} (l_1, \ldots, l'_k, \ldots, l_n, \nu[R])} \text{[TAU]}$$

$$\frac{\nu + \delta \in [\![\bigwedge_{i\in\{i..n\}} I_i(l_i)]\!] \qquad l_1 \notin Loc_1^u \qquad \ldots \qquad l_n \notin Loc_n^u}{(l_1, \ldots, l_n, \nu) \xrightarrow{\delta} (l_1, \ldots, l_n, \nu + \delta)} \text{[DELAY]}$$

$$\frac{\begin{array}{l}(l_h, \texttt{!a}, g_h, R_h, l'_h) \in E_h \\ (l_k, \texttt{?a}, g_k, R_k, l'_k) \in E_k\end{array} \quad \nu \in [\![g_h \wedge g_k]\!] \quad \nu[R_h \cup R_k] \in \begin{array}{l}[\![\bigwedge_{i\in\{1..n\}\setminus\{h,k\}} I_i(l_i)]\!] \\ \cap\, [\![I_h(l'_h)]\!] \cap [\![I_k(l'_k)]\!]\end{array}}{(l_1, \ldots, l_h, \ldots, l_k, \ldots, l_n, \nu) \xrightarrow{\texttt{a}} (l_1, \ldots, l'_h, \ldots, l'_k, \ldots, l_n, \nu[R_h \cup R_k])} \text{[COM]}$$

Figure 2.4: Transition relation of networks of TA.

guard is satisfied, and that the invariants of *all* the locations in the target configuration are satisfied. Rule [DELAY] allows time to elapse, at the same pace for all TA in the network, provided that the invariants at the current locations of all TA are satisfied after the delay. Rule [COM] allows two TA to synchronize on a channel a, provided that the following three conditions hold: (i) at their current locations, the two TA can fire complementary actions (such as !a and ?a); (ii) the clock valuation satisfies the guards of those edges; (iii) the invariants of the target locations are satisfied after the clock reset.

If the current locations of a state have no outgoing edges, then such state is called *success*, while a state is called *deadlock* if it is not success and no *action*-transitions are possible (neither in the current clock valuation, nor in the future).

Definition 2.2.11 (Deadlock freedom)**.** A location of a TA is called success when it has not outgoing edges. Let s be a network state. Then, s is called success when all its locations are success. We say that s is *deadlock* whenever: (i) s is not *success*, and (ii) $\nexists \delta \geq 0,\ \texttt{a}_\tau \in \texttt{A} \cup \{\tau\} : s \xrightarrow{\delta}_N \xrightarrow{\texttt{a}_\tau}_N$. A network is *deadlock-free* if none of its reachable states is deadlock.

Example 2.2.1 (Light switch)**.** Consider a light switch with three modes: off, low and bright. When the light is off and the button is pressed once, the light switches on. Then, if the button is pressed again quickly (in less than 3 seconds), the light becomes brighter. Otherwise, if the delay is greater, the light switches off. Finally, when pressing the button in bright mode, the light switches off. We model this

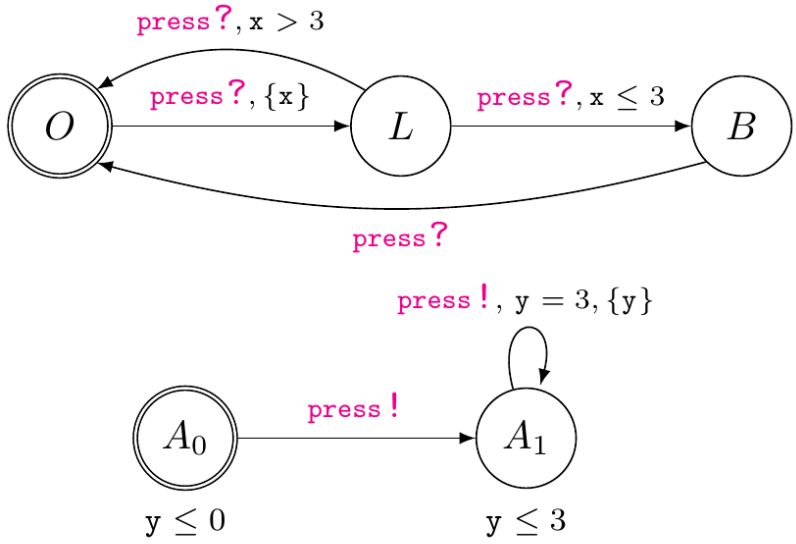


Figure 2.5: A network of TAs modelling a light switch (left) and a user (right).

light switch as the TA in Figure 2.5, left, while on the right we model a possible user. Our user starts by pressing the light switch, and then she repeatedly presses it every 3 seconds.

2.3 Timed CO_2

In this section we introduce TCO_2 (for *timed* CO_2), a specification language for contract-oriented services that we use in our contract-oriented middleware. This is a timed extension of the process calculus in [27], through which we can specify services interacting through primitives analogous to those sketched in Section 3.1.

The formalisation of TCO_2 is independent from the chosen contract language, as we only pivot on a few abstract operators and relations on contracts, although the timed session types described in previous Section 2.1 perfectly fit the requirements of TCO_2. In particular, we assume: (1) a *compliance relation* $\bowtie$, which relates two contracts whenever their interaction is "correct" [21]; (2) a predicate which says if a contract admits a compliant one; (3) a function $\mathrm{co}(\cdot)$ that, given a contract p, gives a contract compliant with p (when this exists); (4) a transition relation $\twoheadrightarrow$ between *contract configurations* γ, γ', which makes contracts evolve upon actions and time passing. We denote with $\Gamma_0(\mathsf{A} : p, \mathsf{B} : q)$ the initial configuration of an interaction between A (with contract p) and B (with contract q).

The syntax of TCO_2 is defined as follows.

2.3.1 Syntax

Let $\mathcal{V}$ and $\mathcal{N}$ be disjoint sets of *variables* (ranged over by $x, y, \ldots$) and *names* (ranged over by $s, t, \ldots$). We use $\vec{u}, \vec{v}, \ldots$ to range over sequences of variables.

Definition 2.3.1. The syntax of TCO_2 is defined as follows:

$$
\begin{array}{llllllllll}
S & ::= & \mathbf{0} & \mid & \mathsf{A}[P] & \mid & s[\gamma] & \mid & (u)S & \mid & S \mid S & \mid & \{\downarrow_u p\}_{\mathsf{A}} \\
P & ::= & \mathbf{0} & \mid & \mathrm{X}(\vec{u}) & \mid & \pi \,.\, P & \mid & (u)P & \mid & u \rhd \{\mathsf{a}_i \,.\, P_i\}_{i \in I} & & \\
\pi & ::= & \tau & \mid & \mathtt{tell} \downarrow_u p & \mid & \mathtt{send}_u\, \mathsf{a} & \mid & \mathtt{idle}(\delta) & \mid & \mathtt{accept}(x) & \mid & \bar{x}y \mid x(y)
\end{array}
$$

If $\vec{u} = u_0, \ldots, u_n$, we will use $(\vec{u})S$ and $(\vec{u})P$ as shorthands for $(u_0) \cdots (u_n)S$ and $(u_0) \cdots (u_n)P$, respectively. We also assume the following syntactic constraints on processes and systems:

1. each occurrence of named processes is prefix-guarded;

2. in $(\vec{u})(\mathsf{A}[P] \mid \mathsf{B}[Q] \mid \cdots)$, it must be $\mathsf{A} \neq \mathsf{B}$;
3. in $(\vec{u})(s[\gamma] \mid t[\gamma'] \mid \cdots)$, it must be $s \neq t$.
4. each variable used in contract primitives can not be used as input/output channel (and vice-versa).

commutative monoidal laws for $\mid$ on processes and systems

$$\mathsf{A}[(v)P] \equiv (v)\,\mathsf{A}[P] \qquad Z \mid (u)Z' \equiv (u)(Z \mid Z') \;\text{ if } u \notin \mathtt{fv}(Z) \cup \mathrm{fn}(Z)$$

$$(u)(v)Z \equiv (v)(u)Z \qquad (u)Z \equiv Z \;\text{ if } u \notin \mathtt{fv}(Z) \cup \mathrm{fn}(Z)$$

$$\{\downarrow_s p\}_{\mathsf{A}} \equiv \mathbf{0} \qquad \mathtt{idle}(0).P \equiv P$$

Figure 2.6: Structural equivalence for CO_2 (Z, Z' range over systems or processes).

Systems $S, S', \ldots$ are the parallel composition of *participants* $\mathsf{A}[P]$, *sessions* $s[\gamma]$, *delimited systems* $(u)S$, and *latent contracts* $\{\downarrow_u p\}_{\mathsf{A}}$. A latent contract $\{\downarrow_x p\}_{\mathsf{A}}$ represents a contract p (advertised by A) which has not been stipulated yet; upon stipulation, the variable x will be instantiated to a fresh session name.

Processes $P, Q, \ldots$ are prefix-guarded processes; the branching construct $u \rhd \{\mathtt{a}_i.\, P_i\}$, which waits for input one of the atoms $\mathtt{a}_i$ and the behave as the corresponding P_i; named processes $\mathrm{X}(\vec{u})$ (used e.g. to specify recursive behaviours); delimited processes $(u)P$; and the nil process $\mathbf{0}$.

Prefixes π include contract advertisement $\mathtt{tell}{\downarrow_u}\, p$, contractual output $\mathtt{send}_u\, \mathtt{a}$, delay $\mathtt{idle}(\delta)$, the accept primitive $\mathtt{accept}(x)$, and channel input/output $\bar{x}y$ and $x(y)$. In each prefix $\pi \neq \tau$, the index u refers to the target session involved in the execution of π.

The only binder for names is the delimitation (u), both in systems and processes. Instead, variables have two binders: delimitations (x) (both in systems and processes), and input actions. We stipulate that each process identifier X has a unique defining equation $\mathrm{X}(x_1, \ldots, x_j) \stackrel{\text{def}}{=} P$ such that $\mathtt{fv}(P) \subseteq \{x_1, \ldots, x_j\} \subseteq \mathcal{V}$. We will sometimes omit the arguments of $\mathrm{X}(\vec{u})$ when they are clear from the context. As usual, we omit trailing occurrences of $\mathbf{0}$ in processes.

2.3.2 Semantics

The semantics of TCO_2 is summarised in Figure 2.7 as a reduction relation between systems. The labels are used to separate urgent actions from non-urgent ones.

Definition 2.3.2. The semantics of TCO_2 is the least relation closed under the rules in Figure 2.7.

When an urgent label is enabled, time is not allowed to pass (similarly to the `asap` operator in U-LOTOS [76]). This enforces a fairness property: if an urgent action is enabled, the scheduler can not prevent it by letting time pass. In TCO_2, every discrete action is urgent, except for `fuse`; this formalises the intuition that a session between two compliant contracts can be created at any time by the middleware, independently from the participants' behaviour.

Rule [Tell] adds to the system a latent contract $\{\downarrow_u p\}_{\mathsf{A}}$, if p admits a compliant contract. Rule [Fuse] searches the system for compliant pairs of latent contracts, i.e. $\{\downarrow_x p\}_{\mathsf{A}}$ and $\{\downarrow_y q\}_{\mathsf{B}}$ such that $p \bowtie q$ (and $\mathsf{A} \neq \mathsf{B}$). Then, a fresh session s containing the initial configuration $\gamma = \Gamma_0(\mathsf{A} : p, \mathsf{B} : q)$ is established, and the name s is shared between A and B. Rule [Acpt] allows A to accept a latent contract q, which is passed through the channel x; then, the contract of A at s will be $\text{co}(q)$.

Rule [Send] allows A to send a message `!a` to the other endpoint of session s. This is only permitted if the contract configuration at s can take a transition on $\mathsf{A} : \texttt{!a}$, whereas messages not conforming to the contract will make A culpable of a violation. Rule [Recv] allows A to receive a message $\mathtt{a}_j$ from the other endpoint of s, and to behave like the continuation P_j. Rule [Delay-γ] allows a session $s[\gamma]$ to idle, if permitted by the contract configuration γ at s (note that idling may make one of the participants culpable). Rule [Idle] is standard [76], and it allows a process to idle for a certain time δ.

Rules [Def], [Par-Act] and [Del] are mostly standard. Rule [Delay-K] allows a latent contract to idle indefinitely. Rules [Delay-P] model time elapsing for processes without timed constructs at the top level of the syntax. Roughly, time passes only when urgent actions are not possible. Rule [Delay-Par] allows parallel composition of systems to idle if all the components can, and no urgent transitions are possible.

$$\frac{\exists q : p \bowtie q}{\mathsf{A}[\mathtt{tell} \downarrow_u p.\ P] \xrightarrow{\mathtt{tell}} \mathsf{A}[P] \mid \{\downarrow_u p\}_\mathsf{A}} \text{[Tell]}$$

$$\frac{p \bowtie q \qquad \gamma = \mathsf{A} : p, \nu_0 \mid \mathsf{B} : q, \eta_0 \qquad \sigma = \{^s/_{x,y}\} \qquad s \text{ fresh}}{(x,y)(S \mid \{\downarrow_x p\}_\mathsf{A} \mid \{\downarrow_y q\}_\mathsf{B}) \xrightarrow{\mathtt{fuse}} (s)(S\sigma \mid s[\gamma])} \text{[Fuse]}$$

$$\frac{\gamma = \mathsf{A} : \mathrm{co}(p), \nu_0 \mid \mathsf{B} : p, \nu_0 \qquad \sigma = \{^s/_x\} \qquad s \text{ fresh}}{(x)(\mathsf{A}[\mathtt{accept}(x).\ P] \mid \{\downarrow_x p\}_\mathsf{B} \mid S) \xrightarrow{\mathtt{accept}} (s)(\mathsf{A}[P\sigma] \mid s[\gamma] \mid S\sigma)} \text{[Acpt]}$$

$$\frac{\gamma \xrightarrow{\mathsf{A}:!\mathsf{a}} \gamma'}{\mathsf{A}[\mathtt{send}_s\, \mathsf{a}.\ P] \mid s[\gamma] \xrightarrow{\mathtt{send}} \mathsf{A}[P] \mid s[\gamma']} \text{[Send]}$$

$$\frac{\gamma \xrightarrow{\mathsf{A}:?\mathsf{a}} \gamma' \qquad \mathsf{a} = \mathsf{a}_j}{\mathsf{A}[s \rhd \{\mathsf{a}_i.\ P_i\}] \mid s[\gamma] \xrightarrow{\mathtt{receive}} \mathsf{A}[P_j] \mid s[\gamma']} \text{[Recv]}$$

$$\frac{\mathrm{X}(\vec{x}) \stackrel{\text{def}}{=} P \qquad \mathsf{A}[P\{\vec{u}/\vec{x}\}] \mid S \xrightarrow{\mu} S'}{\mathsf{A}[\mathrm{X}(\vec{u})] \mid S \xrightarrow{\mu} S'} \text{[Def]} \qquad \frac{S \xrightarrow{\mu} S' \qquad \mu \neq \delta}{S \mid S'' \xrightarrow{\mu} S' \mid S''} \text{[Par-Act]}$$

$$\frac{S \xrightarrow{\mu} S'}{(u)S \xrightarrow{\mu} (u)S'} \text{[Del]} \qquad \{\downarrow_u p\}_\mathsf{A} \xrightarrow{\delta} \{\downarrow_u p\}_\mathsf{A} \text{[Delay-K]} \qquad \frac{\gamma \xrightarrow{\delta} \gamma'}{s[\gamma] \xrightarrow{\delta} s[\gamma']} \text{[Delay-}\gamma\text{]}$$

$$\frac{P \neq \mathtt{idle}(\delta').\ P' \qquad \forall \mu \in \mathrm{Urg} : \left(\mathsf{A}[P] \overset{\mu}{\not\to} \ \wedge\ \forall \delta' \leq \delta :\ \mathsf{A}[P] \xrightarrow{\delta'} S \implies S \overset{\mu}{\not\to}\right)}{\mathsf{A}[P] \xrightarrow{\delta} \mathsf{A}[P]} \text{[Delay-P]}$$

$$\frac{\delta' \leq \delta}{\mathsf{A}[\mathtt{idle}(\delta).\ P] \xrightarrow{\delta'} \mathsf{A}[\mathtt{idle}(\delta - \delta').\ P]} \text{[Idle]}$$

$$\frac{S_0 \xrightarrow{\delta} S_0' \qquad S_1 \xrightarrow{\delta} S_1' \qquad \forall \mu \in \mathrm{Urg} : \left(S_0 \mid S_1 \overset{\mu}{\not\to} \ \wedge\ \forall \delta' \leq \delta :\ S_0 \mid S_1 \xrightarrow{\delta'} S \implies S \overset{\mu}{\not\to}\right)}{S_0 \mid S_1 \xrightarrow{\delta} S_0' \mid S_1'} \text{[Delay-Par]}$$

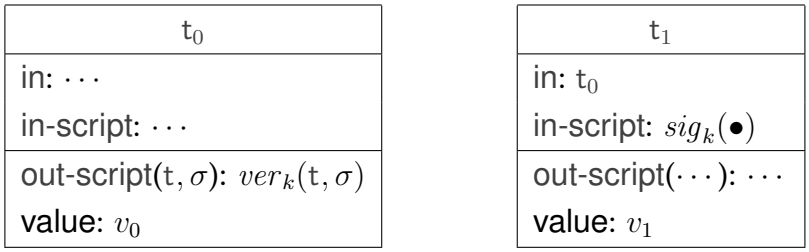
$$\mathsf{A}[\bar{x}y.\ P] \mid \mathsf{B}[x(z).\ Q] \xrightarrow{\tau} \mathsf{A}[P] \mid \mathsf{B}[Q\{^y/_z\}] \text{ [Comm]}$$

Figure 2.7: Reduction semantics of TCO_2.

2.4 Bitcoin and the blockchain

Bitcoin is a cryptocurrency and a digital open-source payment infrastructure that has recently reached a market capitalization of almost $120 billions[3]. The Bitcoin network is peer-to-peer, not controlled by any central authority [73]. Each Bitcoin user owns one or more personal wallets, which consist of pairs of asymmetric cryptographic keys: the public key uniquely identifies the user *address*, while the private key is used to authorize payments. *Transactions* describe transfers of bitcoins (฿), and the history of all transactions, which recorded on a public, immutable and decentralised data structure called *blockchain*, determines how many bitcoins are contained in each address.

To explain how Bitcoin works, we consider two transactions $\mathtt{t}_0$ and $\mathtt{t}_1$, which we graphically represent as follows:[4]



The transaction $\mathtt{t}_0$ contains v_0฿, which can be *redeemed* by putting on the blockchain a transaction (e.g., $\mathtt{t}_1$), whose in field is the cryptographic hash of the whole $\mathtt{t}_0$ (for simplicity, just displayed as $\mathtt{t}_0$ in the figure). To redeem $\mathtt{t}_0$, the in-script of $\mathtt{t}_1$ must contain values making the out-script of $\mathtt{t}_0$ (a boolean programmable function) evaluate to true. When this happens, the value of $\mathtt{t}_0$ is transferred to the new transaction $\mathtt{t}_1$, and $\mathtt{t}_0$ is no longer redeemable. Similarly, a new transaction can then redeem $\mathtt{t}_1$ by satisfying its out-script.

In the example displayed above, the out-script of $\mathtt{t}_0$ evaluates to true when receiving a digital signature σ on the redeeming transaction $\mathtt{t}$, with a given key pair k. We denote with $ver_k(\mathtt{t}, \sigma)$ the signature verification, and with $sig_k(\bullet)$ the signa-

[3] Source: crypto-currency market capitalizations http://coinmarketcap.com

[4] in-script and out-script are respectively referred as scriptPubKey and scriptSig in the Bitcoin documentation.

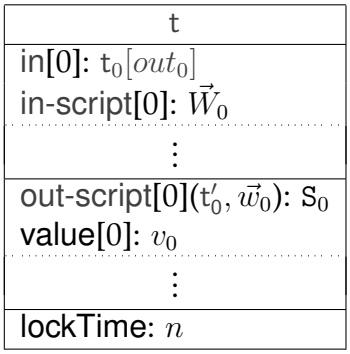


Figure 2.8: General form of transactions.

ture of the enclosing transaction ($\mathtt{t}_1$ in our example), including *all* the parts of the transaction *except* its in-script.

Now, assume that the blockchain contains $\mathtt{t}_0$, not yet redeemed, when someone tries to append $\mathtt{t}_1$. To validate this operation, the nodes of the Bitcoin network check that $v_1 \leq v_0$, and then they evaluate the out-script of $\mathtt{t}_0$, by instantiating its formal parameters $\mathtt{t}$ and σ, to $\mathtt{t}_1$ and to the signature $sig_k(\bullet)$, respectively. The function ver_k verifies that the signature is correct: therefore, the out-script succeeds, and $\mathtt{t}_1$ redeems $\mathtt{t}_0$.

Bitcoin transactions may be more general than the ones illustrated by the previous example: their general form is displayed in Figure 2.8. First, there can be multiple inputs and outputs (denoted with array notation in the figure). Each output has an associated out-script and value, and can be redeemed independently from others. Consequently, in fields must specify which output they are redeeming ($\mathtt{t}_0[out_0]$ in the figure). Similarly, a transaction with multiple inputs associates an in-script to each of them. To be valid, the sum of the values of all the inputs must be greater or equal to the sum of the values of all outputs. In its general form, the out-script is a program in a (not Turing-complete) scripting language, featuring a limited set of logic, arithmetic, and cryptographic operators. Finally, the lockTime field specifies the earliest moment in time (block number or UNIX timestamp) when the transaction can appear on the blockchain.

The Bitcoin network is populated by a large set nodes, called *miners*, which collect transactions from clients, and are in charge of appending the valid ones to the blockchain. To this purpose, each miner keeps a local copy of the blockchain, and a set of unconfirmed transactions received by clients, which it groups into *blocks*.

The goal of miners is to add these blocks to the blockchain, in order to get a revenue. Appending a new block B_i to the blockchain requires miners to solve a cryptographic puzzle, which involves the hash $h(B_{i-1})$ of block B_{i-1}, a sequence of unconfirmed transactions $\langle T_i \rangle_i$, and some salt R. More precisely, miners have to find a value of R such $h(h(B_{i-1}) \parallel \langle T_i \rangle_i \parallel R) < \mu$, where the value μ is adjusted dynamically, depending on the current hashing power of the network, to ensure that the average mining rate is of 1 block every 10 minutes.

The goal of miners is to win the "lottery" for publishing the next block, i.e. to solve the cryptopuzzle before the others; when this happens, the miner receives a reward in newly generated bitcoins, and a small fee for each transaction included in the mined block. If a miner claims the solution of the current cryptopuzzle, the others discard their attempts, update their local copies of the blockchain with the new block B_i, and start mining a new block on top of B_i. In addition, miners are asked to verify the validity of the transactions in B_i by executing the associated scripts. Although verifying transactions is not mandatory, miners are incentivized to do that, because if in any moment a transaction is found invalid, they lose the fee earned when the transaction was published in the blockchain.

If two or more miners solve a cryptopuzzle simultaneously, they create a *fork* in the blockchain (i.e., two or more parallel valid branches). In the presence of a fork, miners must choose a branch wherein carrying out the mining process; roughly, this divergence is resolved once one of the branches becomes longer than the others. When this happens, the other branches are discarded, and all the orphan transactions contained therein are nullified.

Overall, this protocol essentially implements a *"Proof-of-Work"* system [51].

Chapter 3

A contract-oriented middleware

In what follows, we present our centralized middleware for coordinating mutually distrusted service through behavioural contracts. In this chapter, the original contribution of the author are the middleware, with its design and architecture and its implementation, the validating experiments and the contract-oriented programming tutorial.

3.1 The middleware at a glance

Figure 3.1 illustrates the main features of this middleware. In (1), the participant A advertises its contract to the middleware, making it available to other participants.

In (2), the middleware determines that the contracts of A and B are compliant, and then it establishes a session through which the two participants can interact. This interaction consists in sending and receiving messages, similarly to a standard MOM [15]: for instance, in (3) participant A delivers to the middleware a message for B, which can then collect it from the middleware.

Unlike standard MOMs, the interaction happening in each session is monitored by the middleware, which checks whether contracts are respected or not. In particular, the execution monitor verifies that actions can only occur when prescribed by their contracts, and it detects when some expected action is missing. For in-

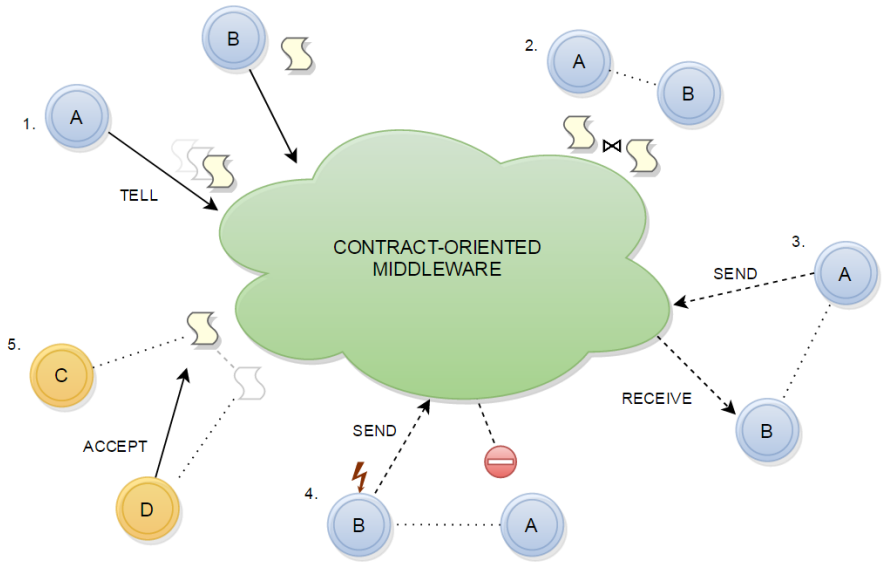


Figure 3.1: A schema of the primitive behaviours.

stance, in (4) the execution monitor has detected an attempt of participant B to do some illegal action. Upon detection of a contract violation, the middleware punishes the culprit, by suitably decreasing its *reputation*. This is a measure of the trustworthiness of a participant in its past interactions: the lower is the reputation, the lower is the probability of being able to establish new sessions with it. The reputation system exploits some of the techniques in [83] to mitigate self-promoting attacks [59].

Item (5) shows another mechanism for establishing sessions: here, the participant C advertises a contract, and D just *accepts* it. Technically, this requires the middleware to construct the *dual* of the contract of C, to associate it with D, and to establish a session between C and D. The interaction happening in this session then proceeds as described previously.

We implemented the primitives discussed above as public REST APIs: their functionalities are partitioned over different architectural components (organized by logical domains), in order to correctly and consistently manage the data flow of the middleware. These components are discussed in the following Section 3.2.

3.2 System design

Hereafter, we show how the interaction paradigm sketched in Section 3.1 (and formalised in Section 2.3) is supported by our middleware, and we illustrate the main design choices.

3.2.1 Specifying contracts

Although the design of the middleware is mostly contract-agnostic, in this thesis we describe and evaluate timed session types (TSTs) as a particular instance of contracts. We now recall some notions about timed session types from Section 2.1, and refer to that section for a full technical development.

Clocks $x, y, \ldots$ are variables over $\mathbb{R}_{\geq 0}$, which can be reset, and used within *guards* $g, g', \ldots$. Atomic guards are timing constraints of the form $x \circ d$ or $x - y \circ d$, where $d \in \mathbb{N}$ and $\circ \in \{<, \leq, =, \geq, >\}$, and they can be composed with the boolean connectives $\wedge$, $\vee$, $\neg$.

A TST p describes the behaviour of a single participant involved in an interaction. An *internal choice* $\sum_i \mathtt{!a}_i\{g_i, R_i\}\,.\,p_i$ models the fact that its participant wants to do one of the outputs with label $\mathtt{a}_i$ in a time window where the guard g_i is true; the clocks in R_i will be reset after the output is performed. An *external choice* $\&_i \mathtt{?a}_i\{g_i, R_i\}\,.\,q_i$ models the fact that its participant is available to receive each message $\mathtt{a}_i$ at *any instant* within the time window where the guard g_i is true; furthermore, the clocks in R_i will be reset after the input is received. The term $\mathbf{1}$ denotes *success* (i.e., a terminated interaction). Infinite behaviour can be specified through recursion $\mathrm{rec}\, X\,.\,p$.

Timed session types $p, q, \ldots$ are terms of the following grammar:

$$p \;::=\; \mathbf{1} \;\mid\; \textstyle\sum_{i\in I} \mathtt{!a}_i\{g_i, R_i\}\,.\,p_i \;\mid\; \&_{i\in I} \mathtt{?a}_i\{g_i, R_i\}\,.\,p_i \;\mid\; \mathrm{rec}\, X\,.\,p \;\mid\; X$$

where

1. the set I is finite and non-empty,
2. the labels in internal/external choices are pairwise distinct,

3. recursion is guarded and considered up-to unfolding.

True guards, empty resets, and trailing occurrences of $\mathbf{1}$ can be omitted.

Message labels are grouped into *contexts*, which can be created and made public through the middleware APIs. Each context defines the labels related to an application domain, and it associates each label with a *type* and a *verification link*. The type (e.g., `int`, `string`) is that of the messages exchanged with that label. The verification link is used by the runtime monitor (Section 3.2.4) to delegate the verification of messages to a trusted third party. For instance, the middleware supports Paypal as a verification link for online payments. The context also specifies the duration of a time unit: the shortest time unit supported by the middleware is that of seconds, which is also the one we use in all the examples in this thesis.

3.2.2 Advertising contracts

Once a contract has been created, a participant can advertise it to the middleware. At that point, the contract stays *latent* until the middleware finds a *compliant* one, i.e. another latent contract with whom the interaction is guaranteed not to get stuck. When this is found, the middleware creates a *session* between the two participants: the session consists of a private channel name and a *contract configuration*, which keeps track of the state of the contract execution.

The notion of compliance between TSTs (Definition 6 in [18]) is based on a transition system over contract configurations (Definition 5 in [18]). Contract configurations have the form $(p, \nu) \mid (q, \eta)$, where p, q are TSTs, and ν, η are *clock evaluations* (i.e., functions from clocks to $\mathbb{R}_{\geq 0}$); in the initial configuration $\Gamma_0(\mathsf{A} : p, \mathsf{B} : q)$, the clock evaluations map each clock to 0. Intuitively, p and q are compliant (in symbols, $p \bowtie q$) if, in all reachable configurations, the "required" behaviour of p (i.e., the branches in its internal choice) is "offered" by q in an external choice, while respecting the time constraints.

Example 3.2.1. Let $p = \mathtt{?a}\{x \leq 2\} \;\&\; \mathtt{?b}\{x \leq 5\}$, and consider the following TSTs:

$$q_1 = \mathtt{!a}\{y \leq 1\} \qquad q_2 = \mathtt{!a}\{y \leq 3\} \qquad q_3 = \mathtt{!a}\{y \leq 2\} + \mathtt{!c}\{y \leq 2\}$$

We have that $p \bowtie q_1$: indeed, q_1 wants to output a within one time unit, and p is available to input a for two time units; compliance follows because the time window for the input includes that for the output. On the contrary, $p \not\bowtie q_2$, since the time window required by q_2 is larger than the one offered by p. Finally, $p \not\bowtie q_3$: although the timing constraints for label a match, q_3 can also choose to send c, which is not among the labels offered by p in its external choice.

Deciding compliance. Compliance between TSTs is decidable (Th. 1 in [18]). To check if $p \bowtie q$, we use the encoding in [18] to translate p and q into Uppaal timed automata [31], and then we model-check the resulting network for deadlock freedom. This amounts to solve the reachability problem for timed automata, whose theoretical worst-case complexity is exponential (more precisely, the problem is PSPACE-complete [5]). In practice, the overall execution time for compliance checking for the TSTs in our test suite is in the order of milliseconds; e.g., in the experimental setup described in Section 4.4, it takes approximately 20ms to check compliance between the largest TSTs on our hand, i.e. those modelling PayPal Protection for Buyers [3]. Since, however, the execution time of compliance checking is non-negligible, we do not perform an exhaustive search when searching the contract store for compliant pairs of contracts; rather, we use the techniques described in the following paragraphs to reduce the search space.

Compliance pre-check. When a TST is advertised, the middleware stores in its database the associated timed automaton (which is then computed only once for each TST), and a *digest* of the TST; this digest comprises its context, and one bit which tells whether its top-level operation is an internal or an external choice (up-to unfolding). When looking for a contract compliant with p, the digests are used to rule out (without invoking the Uppaal model checker) some contracts which are surely *not* compliant with p. In particular, we rule out those q belonging to a context different from that of p, and those with the same top-level operator as p (as internal choices can only be compliant with external ones, and *vice versa*). The remaining contracts are potentially compliant with p, and so we restrict the search space to them. The search also takes into account the reputation of the participants who have advertised these contracts, as described in the following paragraph.

Reputation. The middleware assigns to each participant a *reputation*, which measures its ability to respect contracts. Intuitively, the reputation is increased when the participant successfully completes a session, while it is decreased when it is found culpable of a contract violation (more details about the formulation of the reputation system in Section 3.2.4). Reputation is used to sort latent contracts when searching for compliant pairs: the higher the participant's reputation, the higher the probability to establish a session with it. When looking for a contract compliant with p, we first construct the list of contracts potentially compliant with it (sorted by descending reputation). Then, we randomly choose one of them, according to the folded normal probability distribution. This causes contracts with high reputation to be chosen with high probability, while giving some chances also to contracts with low reputation. If the chosen contract is not compliant with p, it is discarded, and the algorithm chooses another one.

Checking the existence of a compliant. Not all TSTs admit a compliant one. For instance, no contract can be compliant with $p = \texttt{!a}\{y < 7\}.\texttt{?b}\{y < 5\}$, because if p outputs `a` at time 6, the counterpart cannot send `b` in the required time constraint. A sound and complete decision procedure for the existence of a compliant is developed in [18]. When advertising a contract, we use this procedure to rule out those contracts which do not admit a compliant one.

3.2.3 Accepting contracts

As discussed in Section 3.1, a participant A can establish a session with B by accepting one of its contracts, whose identifier has been made public by B. Technically, when A declares to accept a contract p, the middleware constructs the *dual* of p, and assigns it to A. The dual of p is the greatest contract compliant with p, according to the subcontract preorder [18]: intuitively, it is the one whose offers match all of p's requests, and whose requests match all p's offers.

Unlike in the untimed case, the naïve construction of the dual of a TST p (i.e., the one which simply swaps inputs with outputs and internal choices with external ones) does not always produce a compliant TST. For instance, the naïve dual of $p = \texttt{?a}\{x \leq 2\}.\texttt{?b}\{x \leq 1\}$ is $q = \texttt{!a}\{x \leq 2\}.\texttt{!b}\{x \leq 1\}$, which is *not* compliant

with p. Indeed, since q can output !a at any time $1 < \delta \leq 2$, the interaction between p and q can become deadlock, and so they are not compliant.

The dual construction used by the middleware is the one defined in [18], which guarantees to obtain a TST compliant with p, if it exists. Roughly, the construction turns all the internal choices into external ones (without changing guards), and it turns external choices into internal ones, updating the guards to preserve future interactions. For instance, in the example above we obtain the TST !a$\{x \leq 1\}$. !b$\{x \leq 1\}$, which is compliant with p.

3.2.4 Service interaction and runtime monitoring

When a session is established, the participants at the two endpoints can interact by sending and receiving messages. At a more concrete level, sending a message through a session is implemented by posting the message to the middleware, through its RESTful API. The middleware logs the whole interaction history, by recording and timestamping all the messages exchanged in the session. Receiving a message is also implemented by invoking the middleware API; upon a receive request, the middleware inspects the session history to retrieve the first unread message (which is then marked as read). The interaction over the session is asynchronous, as the middleware (similarly to a standard MOM) interprets the session history as two unbounded FIFO buffers containing the messages sent by the two endpoints[1]. However, differently from standard MOMs, our middleware monitors the interaction to verify that contracts are respected.

The runtime monitor processes each message exchanged in a session, by querying the verification link associated to it (to detect whether the message is genuine or not), and by checking that the message is permitted in the current contract configuration. Then, the monitor computes who is in charge of the next move, and, in case of contract violations, it detects which of the two participants is culpable. A participant A can become culpable for different reasons:

1. A sends a message not expected by her contract;
2. A's contract is an internal choice, but A loses time until all the branches

[1] Asynchronous communication is possible despite TSTs having a synchronous semantics, as the middleware is delegated to receive messages on behalf of the recipient.

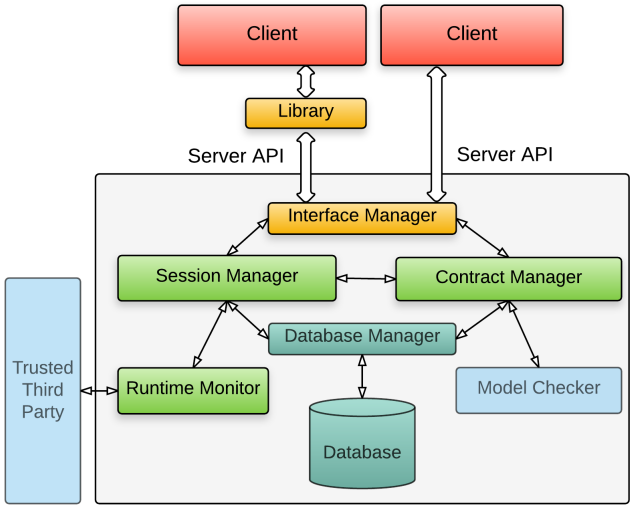


Figure 3.2: A diagram of the middleware architecture.

become unfeasible (i.e., the time constraints are no longer satisfiable);

3. A sends some action at a valid time, but the trusted third party (associated to the action by the verification link) rejects it. For instance, this can happen if A tries to send a fake payment, but Paypal does not certify it.

The monitor guarantees that, in all possible states of the interaction, only one of the participants can be in charge of the next action; if no one is in charge nor culpable, then both participants have reached success (Lemma 3 in [18]).

Once a session terminates (either succesfully or not), the reputation of the involved participants is updated. If the session terminates successfully, then the reputation of both participants is increased; otherwise, the reputation of the culpable participant is decreased, while the other participant's reputation is increased. Further, we make participants consume reputation points each time they enter in session, and we use the *fading memories* technique of [83] to calculate the reputation value without recording the whole history of interactions. We weight recent negative behavior more than old positive behaviour, in order to mitigate *self-promoting attacks*, where a malicious participant tries to gain reputation by running successful sessions with himself or with some accomplices [59].

3.3 System architecture

The middleware is a Java RESTful Web service; the primitives described in Section 3.2 are organised in components, as shown in Figure 3.2. We have adopted a 3-tier architecture, consisting of a presentation layer, a business logic layer, and a data storage layer. The Interface Manager, which is the only component in the presentation layer, offers APIs to query the middleware, through HTTP POST requests. APIs can be accessed through language-specific libraries, which allow for an object-oriented programming style. The data storage layer comprises a relational DB and a Database Manager, which takes care of handling queries, managing the cache, and modelling the data used in the other layers. The business logic layer manages contracts and sessions. More specifically, the Contract Manager performs the contract validation, advertisement (as in Section 3.2.2), and `accept` requests (Section 3.2.3); the Session Manager establishes sessions, by allowing clients to send and receive messages, managing the session history, and querying the Runtime Monitor to detect contract violations.

A client advertises a contract p with the `tellContract` API of the Interface Manager, encoding the required data in the JSON data exchange format. The Interface Manager validates p, then it asks the Contract Manager to store it and to find a compliant contract, as outlined in Section 3.2.2. If no latent contracts are compliant with p, then p is kept latent, otherwise a new session is established. The Interface Manager also provides the `acceptContract` API, which requires the Contract Manager to compute the dual of a latent contract q, whose identifier has been made public by another participant.

When a session is established, participants can query the middleware to get the current time, to send and receive messages, to check culpability, etc. The Interface Manager provides the methods for handling such requests, delegating the internal operations to the Session Manager. When a participant `sends` a message, the Session Manager uses the Runtime Monitor to determine whether the action is permitted (and in case it is not, to assign the blame). If the action is permitted, the message is stored by the Database Manager, and then forwarded to the other participant upon a `receive`. To verify a message, the Runtime Monitor can invoke a trusted third party: if the verification fails, the action is rejected (so, our monitor implements *truncation*, in the terminology of [68]).

3.4 Validation of the middleware

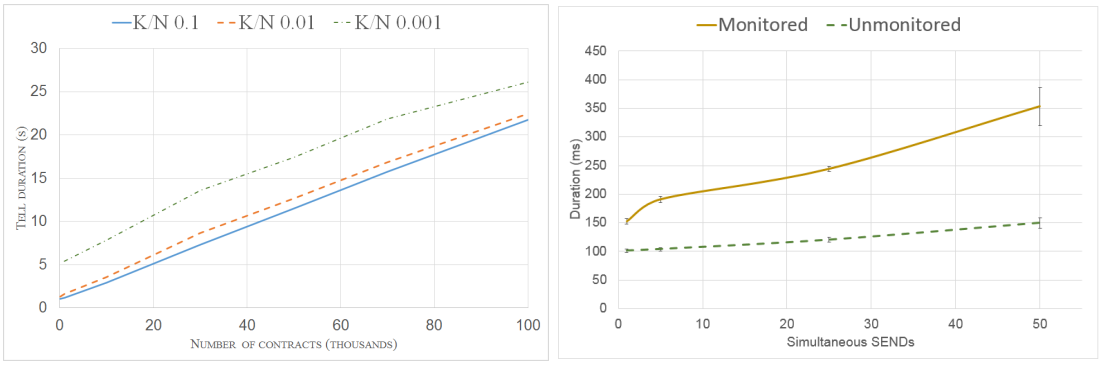
In this section we validate our middleware, mainly focusing on the aspects related to system scalability (Section 3.4.1), and to the effectiveness of the reputation system to rule out services not respecting contracts (Section 3.4.2).

We carry out our experiments using a public instance of the middleware. The instance is a Web service running in a dedicated cloud server, equipped with a quad-core Intel Xeon CPU @ 2.27GHz, 16GB of RAM and a 50GB SSD hard drive; the server runs Ubuntu 14.04 LTS, with Apache Tomcat and Oracle MySQL. Clients are tested in standard desktop PCs and laptops, while the multi-threaded simulations are executed in a high-level desktop configuration, with an octa-core Intel Core i7 @ 4.00GHz and 16Gb of memory, running Microsoft Windows 7 and Oracle JRE 1.7.

3.4.1 Scalability

In this section we assess the scalability of our middleware. We start by benchmarking the `tell` primitive, which triggers a search for compliant pairs of TSTs in the contract store. This is the most computationally expensive operation in the middleware: although the heuristics discussed in Section 3.2.2 allow for limiting the number of calls to the Uppaal model checker, the execution time of a `tell` could be non-negligible for a high number of latent contracts. So, we measure the execution time of `tell` p as a function of the number of TSTs in the contract store, and of the number of latent TSTs compliant with p.

Our second experiment concerns the performance of the runtime monitor. As described in Section 3.2.4, this component processes all the messages exchanged in sessions, to check if contracts are respected. Potentially, this could introduce a relevant computational overhead, so we measure the execution time of `send` in case the runtime monitor is turned on, or off. Note that, while the duration of `tell` does not affect the interaction between the participants once a session is established, a slowdown of the `send` can make an otherwise-honest participant culpable for not respecting some deadline. So, it is important that the overhead of the runtime monitor is negligible, w.r.t. the time scale of temporal constraints.



(a) Duration of `tell` p (in seconds). (b) Duration of `send` (in milliseconds).

Figure 3.3: Results of the scalability tests. In (a), K is the number of contracts compliant with p, and N is the total number of contracts.

We build our scalability tests upon the discrete-event simulator DESMO-J [55], and the statistical model-checker MultiVeStA [82]. In particular, we use DESMO-J to define a single instance of the simulation, and MultiVeStA to run sequences of simulations until reaching a given confidence interval.

Tell. We test the execution time of `tell` p as a function of the number N of contracts stored in the middleware. The contract p used in our experiments is a simplified version of the Paypal Protection for Buyers (Example 1 in [18]). We assume that, among the N contracts, only $K \ll N$ are compliant with p, while the remaining $N - K$ are not, but they still pass the pre-check discussed in Section 3.2.2 (so, we are considering a worst-case scenario, because in the average case we expect that only a fraction of the contracts would pass the pre-check).

We populate the contract store by choosing at each step whether to insert a contract compliant with p or a non-compliant one, according to a random weighted probability. Then, with DESMO-J we execute `tell` p, and we measure its execution time. MultiVeStA makes DESMO-J execute this simulation for several times, each time collecting the new `tell` duration and updating the average and the standard deviation; the simulations stop when the average fits into the confidence interval.

The results of our experiments are shown in Figure 3.3. As we can see, the `tell` duration grows linearly with N, and it increases by a constant when the percentage K/N of contracts compliant with p decreases; note that the slope of the curves does not seem to be significantly affected by K/N.

Runtime monitor. The goal of this experiment is to quantify how the execution of a large number of simultaneous `send` affects the performance of the middleware. To achieve this goal, we use a multi-threaded simulation, where all the threads advertise a contract with an internal sum, wait the session to be established, and then simultaneously perform the `send`. We repeat the measure of the `send` duration until its standard deviation fits into the confidence interval. The results of this experiment are reported in Figure 3.3b, which shows that the execution of a large number of simultaneous `send`s penalizes the duration of the request, compared to the situation where the runtime monitor is switched off. However, the performance degradation seem to grow sub-linearly in the number of simultaneous requests, and in any case it is negligible w.r.t. the time scale of temporal constraints (1 time unit = 1 second).

3.4.2 A distributed experiment: RSA cracking

Consider a service (hereafter referred to as *master*, or just M) who wants to solve a cryptographic problem by exploiting the computational resources of external nodes (hereafter called *workers*, or W) distributed over the network. In particular, M wants to crack a set of public RSA keys, in order to get the corresponding private keys. However, the master does not know the network structure (i.e., how many workers are available, where they are located, and how they are connected), and it does not have any pre-shared channel for communicating with them. Furthermore, the master does not trust the workers: they are not bound to run any particular cracking algorithm, they can return wrong/incomplete results, or they can fail to answer within the expected deadline.

To cope with these issues, the master exploits our middleware to automatically discover and invoke suitable workers. For each public key in its set, the master spawns a process which advertises the contract:

$$p_{\mathsf{M}} \;=\; \texttt{!pubkey}\{;x\}.\,(\texttt{?confirm}\{x<15\}.\,\texttt{?result}\{x<90\}.\,\texttt{!pay1xbt}\{x<120\} \;\&\; \texttt{?abort}\{x<15\})$$

Here, M is promising to send a public key (`pubkey`); doing so triggers a reset of the clock x. Then, the worker has 15 seconds to either `confirm` that he will carry on

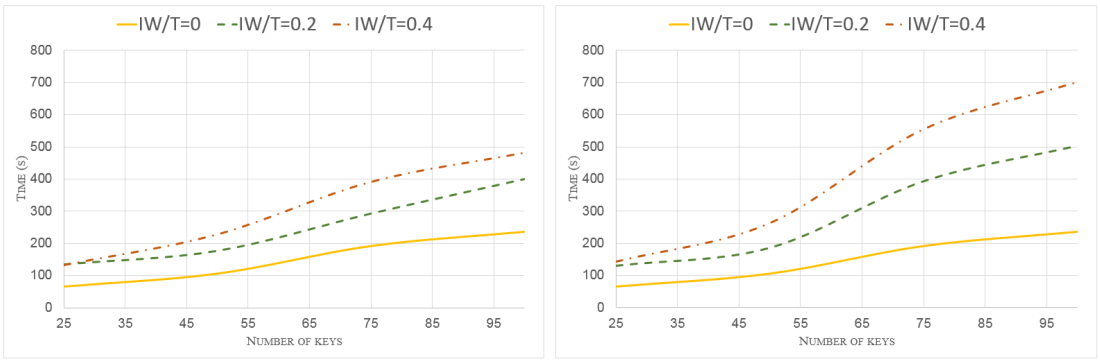
the task, or `abort` (e.g., if the key is considered too strong). If the worker confirms, it must return the corresponding `result` (a private key) within 90 seconds since the public key was sent (the correctness of the result is checked by a trusted third party,[2] specified by the context of p_W); finally, M rewards the worker with 1 bitcoin (`pay1xbt`). At runtime, the master behaves as prescribed by its contract; if the worker accepts the public key and it returns the corresponding private key, then M removes that public key from the list; otherwise, it advertises another instance of p_M, and when the session is established it sends the same public key to another worker.

The advantage offered by the middleware in terms of code succinctness is clear, as the search of workers, the establishment of sessions, and the runtime monitoring is completely transparent to programmers. So, we assess below the reputation system implemented in the middleware (Sections 3.2.2 and 3.2.4). In particular, we measure the time taken by the master for cracking all the public keys in its list (*Overall Execution Time, OET*). We do this in two configurations of the middleware: the one where the reputation system is turned on, and the one where it is turned off. Our conjecture is that turning the reputation system on will reduce the OET, because it increases the probability of establishing sessions with *honest* workers which produce correct results while respecting deadlines.

In our experiments, we assume that workers are drawn from two different classes: those using an efficient cracking algorithm, which always return the correct result within the deadline; and those using an inefficient algorithm, which sometimes may miss the deadline, because the computation takes too long. We also assume that the number of public keys is bigger than the number of workers, so each of them may receive many keys to break. Each worker iteratively advertises its contract (the dual of p_M), then waits for a public key, runs the cracking algorithm, and finally return the private key to the master.

The results of our experiment are shown in Figure 3.4, where we measure the OET as a function of the number of keys to be broken, and of the ratio between efficient and inefficient workers. The solid curve is identical in the two figures, since the reputation system does not affect the selection of workers when there are

[2]Note that verifying the correctness of private keys has a polynomial complexity in the number of bits of the public key, while the problem of cracking RSA keys is considered to be exponentially hard.



(a) OET with reputation enabled.

(b) OET with reputation disabled.

Figure 3.4: Overall Execution Time as a function of the number of keys to be broken. IW is the number of inefficient workers, and T is the total number of workers.

only efficient ones. In the dashed curve and in the dot-dashed one the percentage of inefficient workers grows (to 20% and 40%, respectively), and we see that the OET grows accordingly when the reputation system is turned off. This is because the reputation system penalizes inefficient workers, by reducing the probability they can establish sessions with the master.

3.5 Contract-oriented programming

In this section we show how to develop some simple contract-oriented services, focusing on timed session types only, and using the middleware APIs via their Java binding[3].

Before giving practical examples, we first illustrate an informal version of the TST syntax shown in Section 2.1, with the help of a small case study, an online store which receives orders from customers. Note that the use of *un*timed session types in contract-oriented applications is anyway discussed in the literature [9, 11, 23].

3.5.1 Specifying contracts in practice

Timed session types extend binary session types [60, 86] with clocks and timing constraints, similarly to the way timed automata [5] extend (classic) finite state automata. We informally describe the syntax of TSTs below, and we refer to Section 2.1 for the full technical development.

Guards. Guards describe timing constraints, and they are conjunctions of simple guards of the form `t` $\circ$ `d`, where `t` is a *clock*, `d` $\in \mathbb{N}$, and $\circ$ is a relation in `<`, `<=`, `=`, `>=`, `>`. For instance, the guard `t<60,u>10` is true whenever the value of clock `t` is less than 60, *and* the value of clock `u` is greater than 10. The value of clocks is in $\mathbb{R}_{\geq 0}$, like for timed automata.

Send and receive. A TST describes the behaviour of a single participant A at the end-point of a session. Participants can perform two kinds of actions:

- a *send action* `!a{g;t1,...,tk}` stipulates that A will output a message with label `a` in a time window where the guard `g` is true. The clocks `t1,...,tk` will be reset after the output is performed.
- a *receive action* `?a{g;t1,...,tk}` stipulates that A will be available to receive a message with label `a` at *any instant* within the time window where the guard `g` is true. The clocks `t1,...,tk` will be reset after the input is received.

[3]Full code listings are available at co2.unica.it

When `g = true`, the guard can be omitted.

For instance, consider the contract `store1` between the store and a customer, from the point of view of the store.

```
store1 = "?order{;t} . !price{t<60}"
```

The store declares that it will receive an order at any time. After it has been received, the store will send the corresponding price within 60 seconds.

Internal and external choices. TSTs also feature two forms of choice:

- `!a1{g1;R1} + ... + !an{gn;Rn}`

  This is an *internal choice*, stipulating that A will decide at run-time which one of the output actions `!ai{gi;Ri}` (with $1 \leq i \leq n$) to perform, and at which time instant. After the action is performed, all clocks in the set `Ri = {t1,...,tk}` are reset.

- `?a1{g1;R1} & ...& ?an{gn;Rn}`

  This is an *external choice*, stipulating that A will be able to receive any of the inputs `!ai{gi;Ri}`, in the declared time windows. The actual choice of the action, and of the instant when it is performed, will be made by the participant at the other endpoint of the session. After the action is performed, all clocks in the set `Ri = {t1,...,tk}` are reset.

With these ingredients, we can refine the contract of our store as follows:

```
store2 = "?order{;t} . (!price{t<60} + !unavailable{t<10})"
```

This version of the contract deals with the case where the store receives an unknown or invalid product code. In this case, the internal choice allows the store to inform the buyer that the requested item is `unavailable`.

Recursion. The contracts shown so far can only handle a bounded (statically known) number of interactions. We can overcome this limitation by using recursive TSTs. For instance, the contract `store3` below models a store which handles an arbitrary number of orders from a buyer:

```
store3 = "REC 'x' [?addtocart{t<60;t}.'x'
                 & ?checkout{t<60;t}.(
                       !price{t<20;t}.(
                           ?accept{t<10} & ?reject{t<10})
                     + !unavailable{t<20})]"
```

The contract `store3` allows buyers to add some item to the cart, or checkout. When a buyer chooses `addtocart`, the store must allow him to add more items: this is done recursively. After a `checkout`, the store must send the overall `price`, or inform the buyer that the requested items are `unavailable`. If the store sends a price, it must expect a response from the buyer, who can either `accept` or `reject` the price.

Context. Action labels are grouped into *contexts*, which can be created and made public through the middleware APIs. Each context defines the labels related to an application domain, and it associates each label with a *type* and a *verification link*. The type (e.g., `int`, `string`) is that of the messages exchanged with that label. The verification link is used by the runtime monitor (described later on in this section) to delegate the verification of messages to a trusted third party. For instance, the middleware supports Paypal as a verification link for online payments.

3.5.2 Compliance and duality in practice

Besides being used to specify the interaction protocols between pairs of services, recall that TSTs feature the following primitives:

- a decidable notion of *compliance* between two TSTs;
- an algorithm to detect if a TST admits a compliant one;
- a computable *canonical compliant* construction.

These primitives are exploited by the CO_2 middleware to establish sessions between services: more specifically, recall also that the middleware only allows interactions between services with compliant contracts. In fact, compliance guarantees that, if *all* services respect *all* their contracts, the overall distributed appli-

cation (obtained by composing the services) will not deadlock.

For instance, recall the simple version of the store contract:

```
store1 = "?order{;t} . !price{t<60}"
```

and consider the following buyer contracts:

```
buyer1 = "!order{;u} . ?price{u<70}"
buyer2 = "!order{;u} . (?price{u<70} & ?unavailable)"
buyer3 = "!order{;u} . (?price{u<30} & ?unavailable)"
buyer4 = "!order{u<20} . ?price{u<70}"
```

We have that:

- `store1` and `buyer1` are compliant: indeed, the time frame where `buyer1` is available to receive `price` is larger than the one where the store can send;
- `store1` and `buyer2` are compliant: although the action `?unavailable` enables a further interaction, this is never chosen by the store `store1`.
- `store1` and `buyer3` are *not* compliant, because the store may choose to send `price` 60 seconds after he got the order, while `buyer2` is only able to receive within 30 seconds.
- `store1` and `buyer4` are *not* compliant. Here the reason is more subtle: assume that the buyer sends the order at time 19: at that point, the store receives the order and resets the clock `t`; after that, the store has 60 seconds more to send `price`. Now, assume that the store chooses to send `price` after 59 seconds (which fits within the declared time window of 60 seconds). The total elapsed time is 19+59=78 seconds, but the buyer is only able to receive before 70 seconds.

We can check if two contracts are compliant through the middleware Java APIs[4]. We show how to do this through the Groovy[5] interactive shell[6].

```
cS1 = new TST(store1)
cS1.isCompliantWith(new TST(buyer1))
```

[4] co2.unica.it/downloads/co2api/

[5] groovy-lang.org/download.html

[6] On Unix-like systems, copy the API's jar in `$HOME/.groovy/lib/`. Then, add `import co2api.*` to `$HOME/.groovy/groovysh.rc`, and run `groovysh`.

```
>>> true
cS1.isCompliantWith(new TST(buyer3))
>>> false
```

Consider now the second version of the store contract:

```
store2 = "?order{;t} . (!price{t<60} + !unavailable{t<10})"
```

The contract `store2` is compliant with the buyer contract `buyer2` discussed before, while it is *not* compliant with:

```
buyer5 = "!order{;u} . (?price{u<90})"
buyer6 = "!order{;u} . (?price{u<90} + ?unavailable{u>5,u<12})"
```

The problem with `buyer5` is that the buyer is only accepting a message labelled `price`, while `store2` can also choose to send `unavailable`. Although this option is present in `buyer6`, the latter contract is not compliant with `store2` as well. In this case the reason is that the time window for receiving `unavailable` does not include that for sending it (recall that the sender can choose any instant satisfying the guard in its output action). To illustrate some less obvious aspects of compliance, consider the following buyer contract:

```
buyer7 = "!order{u<100} . ?price{u<70}"
```

This contract stipulates that the buyer can wait up to 100 seconds for sending an order, and then she can wait until 60 seconds (from the *start* of the session), to receive the price from the store.

Now, assume that some store contract is compliant with `buyer7`. Then, the store must be able to receive the `order` at least until time 100. If the buyer chooses to send the `order` at time 90 (which is allowed by contract `buyer7`), then the store would never be able to send `price` before time 70. Therefore, no contract can be compliant with `buyer7`.

The issue highlighted by the previous example must be dealt with care: if one publishes a service whose contract does not admit a compliant one, then the middleware will never connect that service with others. To check whether a contract admits a compliant one, we can query the middleware APIs:

```
cB7 = new TST(buyer7)
>>> !order{u<100} .  ?price{u<70}

cB7.hasCompliant()
>>> false
```

Recall from Section 3.1 that the CO_2 middleware also allows a service to *accept* another service's contract, as per item (5) in Figure 3.1. E.g., assume that the store has advertised the contract `store2` above. When the buyer uses the primitive `accept`, the middleware associates the buyer with the *canonical compliant* of `store2`, constructed through the method `dualOf`, i.e.:

```
cS2 = new TST(store2)
>>> ?order{;t} .  (!price{t<60} + !unavailable{t<10})

cB2 = cS2.dualOf()
>>> !order{;t} .  (?price{t<60} & ?unavailable{t<10})
```

Intuitively, if a TST admits a compliant one, then its canonical compliant is constructed as follows:

1. output labels `!a` are translated into input labels `?a`, and *vice versa*;
2. internal choices are translated into external choices, and *vice versa*;
3. prefixes and recursive calls are preserved;
4. guards are suitably adjusted in order to ensure compliance.

Consider now the following contract of a store which receives an order and a coupon, and then sends a discounted price to the buyer:

```
store4 = "?order{t<60} . ?coupon{t<30;t} . !price{t<60}"
```

In this case `store4` admits a compliant one, but this cannot be obtained by simply swapping input/output actions and internal/external choices.

```
cS4 = new TST(store4)
cB4 = new TST("!order{t<60} . !coupon{t<30;t} . ?price{t<60})")
cS4.isCompliantWith(cB4)
>>> false
```

Indeed, the canonical compliant construction gives:

```
cB5 = cS4.dualOf()
>>> !order{t<30} .  ?coupon{t<30;t} .  ?price{t<60}
```

3.5.3 Run-time monitoring example

In order to detect (and sanction) contract violations, the CO_2 middleware monitors all the interactions that happen through sessions. The monitor guarantees that, in each reachable configuration, only one participant can be "on duty" (i.e., she has to perform some actions); and if no one is on duty nor culpable, then both participants have reached success. Here we illustrate how runtime monitoring works, by making a store and a buyer interact.

To this purpose, we split the content in two columns: in the left column we show the store behaviour, while in the right column we show the buyer. We assume that both participants call the middleware APIs through the Groovy shell, as shown before. Note that the interaction between the two participants is asynchronous: when needed, we will highlight the points where one of the participants performs a time delay.

Both participants start by creating a connection `co2` with the middleware:

```
usr = "testuser1@gmail.com"
pwd = "testuser1"
co2 = new CO2ServerConnection(usr,
    pwd)
```

```
usr = "testuser2@gmail.com"
pwd = "testuser2"
co2 = new CO2ServerConnection(usr,
    pwd)
```

Then, the participants create their contracts, and advertise them to the middleware through the primitive `tell`. The variables `pS` and `pB` are the handles to the published contracts.

```
cS = new TST(store2)
pS = cS.toPrivate(co2).tell()
```

```
cB = new TST(buyer2)
pB = cB.toPrivate(co2).tell()
```

Now the middleware has two compliant contracts in its collection, hence it can establish a session between the store and the buyer. To obtain a handle to the

session, both participants use the blocking primitive `waitForSession`:

```
sS = pS.waitForSession()
```

```
sB = pB.waitForSession()
```

At this point, participants can query the session to see who is "on duty" (namely, one is on duty if the contract prescribes her to perform the next action), and to check if they have violated the contract:

```
sS.amIOnDuty()
>>> false
sS.amICulpable()
>>> false
```

```
sB.amIOnDuty()
>>> true
sB.amICulpable()
>>> false
```

Note that the first action must be performed by the buyer, who must send the `order`. This is accomplished by the `send` primitive. Dually, the store waits for the receipt of the message, using the `waitForReceive` primitive:

```
msg = sS.waitForReceive()
msg.getStringValue()
>>> 0123
sS.amIOnDuty()
>>> true
```

```
// send at an arbitrary time
sB.send("order", "0123")

sB.amIOnDuty()
>>> false
```

Since there are no time constraints on sending `order`, this action can be successfully performed at any time; once this is done, the `waitForReceive` unlocks the store. The store is now on duty, and it must send `price` within 60 seconds, or `unavailable` within 10 seconds. Now, assume that the store tries to send `unavailable` after the deadline:

```
// wait more than 10 seconds

sS.send("unavailable")
>>> ContractException
```

```
msg = sB.waitForReceive()


>>> ContractViolationException:
"The other participant is
    culpable"
```

On the store's side, the `send` throws a `ContractException`; on the buyer side, the `waitForReceive` throws an exception which reports the violation of the store. At

this point, if the two participants check the state of the session, they find that none of them is still on duty, and that the store is culpable:

```
session.amIOnDuty()
>>> false
session.amICulpable()
>>> true
```

```
session.amIOnDuty()
>>> false
session.amICulpable()
>>> false
```

At this point, the session is terminated, and the reputation of the store is suitably decreased.

Now, using the ingredients we have seen before, we are finally ready to show real-world implementations of contract-oriented services.

3.5.4 A simple store

We start with a basic store service, which advertises the contract `store2`:

```
1  String store2 ="?order{;t}.(!price{t<60} + !unavailable{t<10})";
2  TST c = new TST(store2);

4  CO2ServerConnection co2 =
5    new CO2ServerConnection("testuser@co2.unica.it", "pa55w0rd");
6  Private r = c.toPrivate(co2);
7  Public  p = r.tell();          //advertises the contract store2

9  Session s = p.waitForSession();//blocks until session is created
10 String id = s.waitForReceive().getStringValue();

12 if(isAvailable(id)) { s.send("price", getPrice(id)); }
13 else { s.send("unavailable"); }
```

At lines 1–2, the store constructs a TST `c` for contract `store2`. At lines 4–5, the store connects to the middleware, providing its credentials. At line 6, the `Private` object represents the contract in a state where it has not been advertised to the middleware yet. To advertise the contract, we invoke the `tell` method at line 7. This call returns a `Public` object, modelling a latent contract that can be "fused"

with a compliant one to establish a new session. At line 9, the store waits for a session to be established; the returned `Session` object allows the store to interact with a buyer. At line 10, the store waits for the receipt of a message, containing the code of the product requested by the buyer. At lines 12–13, the store sends the message `price` (with the corresponding value) if the item is available, otherwise it sends `unavailable`.

3.5.5 A simple buyer

We now show a buyer that can interact with the store. This buyer just accepts the already published contract `store2`. The contract is identified by its hash, which is obtained from `Public.getContractID()`.

```
1  CO2ServerConnection co2 = new CO2ServerConnection(...);
2
3  String storeCID = "0x...";
4  Integer desiredPrice = 10;
5
6  Public  p = Public.accept(co2, storeCID, TST.class);
7  Session s = p.waitForSession();
8
9  s.send("order", "11235811");
10
11 try {
12     Message m = s.waitForReceive();
13     switch (m.getLabel()) {
14     case "unavailable": break;
15     case "price":
16         Integer price = Integer.parseInt(m.getStringValue());
17         if (price > desiredPrice) { /*abort the purchase*/ }
18         else { /*proceed with the purchase*/ }
19     }
20 } catch(ContractViolationException e){/*The store is culpable*/}
```

At line 6, the buyer accepts the store's contract, identified by `storeCID`. The call to `Public.accept` returns a `Public` object. At this point a session with the store is

already established, and `waitForSession` just returns the corresponding `Session` object (line 7). Now, the buyer sends the item code (line 9), waits for the store response (line 12), and finally in the `try-catch` statement it handles the messages `price` and `unavailable`.

Note that the `accept` primitive allows a participant to establish sessions with a chosen counterpart; instead, this is not allowed by the `tell` primitive, which can establish a session whenever two contracts are compliant.

3.5.6 A dishonest store

Consider now a more complex store, which relies on external distributors to retrieve items. As before, the store takes an order from the buyer; however, now it invokes an external distributor if the requested item is not in stock. If the distributor can provide the item, then the store confirms the order to the buyer; otherwise, it informs the buyer that the item is unavailable.

Our first attempt to implement this refined store is the following.

```
1  TST cB = new TST(store2);
2  TST cD = new TST("!req{;t}.(?ok{t<10} & ?no{t<10})");

4  Public  pB = cB.toPrivate(co2).tell();
5  Session sB = pB.waitForSession();
6  String  id = sB.waitForReceive().getStringValue();

8  if (isAvailable(id)) { // handled internally
9      sB.send("price", getPrice(id));
10 }
11 else { // handled with a distributor
12     Public  pD = cD.toPrivate(co2).tell();
13     Session sD = pD.waitForSession();

15     sD.send("req", id);
16     Message mD = sD.waitForReceive();

18     switch (mD.getLabel()) {
```

```
19      case "no" : sB.send("unavailable"); break;
20      case "ok" : sB.send("price", getPrice(id)); break;
21    }
22  }
```

At lines 1-2 we construct two TSTs: `cB` for interacting with buyers, and `cD` for interacting with distributors. In `cD`, the store first sends a request for some item to the distributor, and then waits for an `ok` or `no` answer, according to whether the distributor is able to provide the requested item or not. At lines 4-6, the store advertises `cB`, and it waits for a buyer to join the session; then, it receives the order, and checks if the requested item is in stock (line 8). If so, the store sends the price of the item to the buyer (line 9).

If the item is not in stock, the store advertises `cD` to find a distributor (lines 12-13). When a session `sD` is established, the store forwards the item identifier to the distributor (line 15), and then it waits for a reply. If the reply is `no`, the store sends `unavailable` to the buyer, otherwise it sends a `price`.

Note that this implementation of the store is *dishonest*, namely it may violate contracts [28]. This happens in the following two cases:

1. Assume that the store has received the buyer's order, but the requested item is not in stock. Then, the store advertises the contract `cD` to find a distributor. Note that there is no guarantee that the session `sD` will be established within a given deadline, nor that it will be established at all. If more than 60 seconds pass on the `waitForSession` at line 13, the store becomes culpable with respect to the contract `cB`. Indeed, such contract requires the store to perform an action before 60 seconds[7] (10 seconds if the action is `unavailable`).
2. Moreover, if the session `sD` is established in timely fashion, a slow or unresponsive distributor could make the store violate the contract `cB`. For instance, assume that the distributor sends message `no` after nearly 10 seconds. In this case, the store may not have enough time to send `unavailable` to the buyer within 10 seconds, and so it becomes culpable at session `sB`.

We have simulated the scenario described in Item 1, by making the store inter-

[7] Recall that any time constraint should be chosen larger enough to: a) satisfy the contract requirements; and, b) take into account any possible network delay.

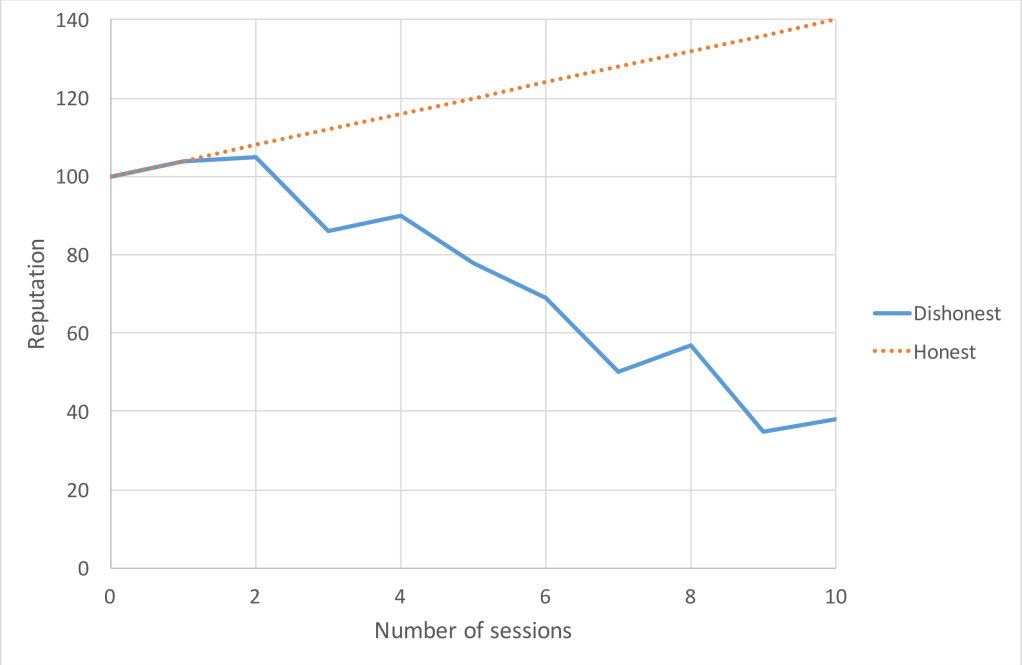


Figure 3.5: Reputation of the dishonest and honest stores as a function of the number of sessions with malicious distributors.

act with slow or unresponsive distributors (see Figure 3.5). The experimental results show that, although the store is not culpable in all the sessions, its reputation decreases over time. Recovering from such situation is not straightforward, since the reputation system of the CO_2 middleware features defensive techniques against self-promoting attacks [83].

3.5.7 An honest store

In order to implement an honest store, we must address the fact that, if the distributor delays its message to the maximum allowed time, the store may not have enough time to respond to the buyer. To cope with this scenario, we adjust the timing constraints in the contract between the store and the distributor, and we implement a revised version of the store as follows.

```
1 TST cB = new TST(store2);
2 TST cD = new TST("!req{;t} . (?ok{t<5} & ?no{t<5})");

4 Public  pB = cB.toPrivate(co2).tell();
```

```
5  Session sB = pB.waitForSession();
6  String  id = sB.waitForReceive().getStringValue();

8  if (isAvailable(id)) { // handled internally
9      sB.send("price", getPrice(id));
10 }
11 else { // handled with the distributor
12     Public pD = cD.toPrivate(co2).tell(3 * 1000);
13     try {
14         Session sD = pD.waitForSession();
15         sD.send("req", id);

17         try{
18             Message mD = sD.waitForReceive();

20             switch (mD.getLabel()) {
21             case "no": sB.send("unavailable"); break;
22             case "ok": sB.send("price", getPrice(id)); break;
23             }
24         } catch(ContractViolationException e){
25             //the distributor did not respect its contract
26             sB.send("unavailable");
27         }
28     } catch(ContractExpiredException e) {
29         //no distributor found
30         sB.send("unavailable");
31     }
32 }
```

The parameter in the `tell` at line 12 specifies a deadline of 3 seconds: if the session `sD` is not established within the deadline, the contract `cD` is retracted from the middleware, and a `ContractExpiredException` is thrown. The store catches the exception at line 28, sending `unavailable` to the buyer.

Instead, if the session `sD` is established, the store forwards the item identifier to the distributor (line 15), and then waits for the receipt of a response from it. If the distributor sends neither `ok` nor `no` within the deadline specified in `cD` (5 seconds),

the middleware assigns the blame to the distributor for a contract breach, and unblocks the `waitForReceive` in the store with a `ContractViolationException` (line 24). In the exception handler, the store fulfils the contract `cB` by sending `unavailable` to the buyer.

3.5.8 A recursive honest store

We now present another version of the store, which uses the recursive contract `store3` on page 50. As in the previous version, if the buyer requests an item that is not in stock, the store resorts to an external distributor.

```
1  TST cB = new TST(store3);
2  TST cD = new TST("!req{;t}.(?ok{t<5} & ?no{t<5})");

4  Public  pB = cB.toPrivate(co2).tell();
5  Session sB = pB.waitForSession();
6  List<String> orders = new ArrayList<>();
7  Message mB;

9  try {
10     do {
11         mB = sB.waitForReceive();
12         if (mB.getLabel().equals("addtocart")){
13             orders.add(mB.getStringValue());
14         }
15     } while(!mB.getLabel().equals("checkout"));

17     if (isAvailable(orders)) { // handled internally
18         sB.send("price", getPrice(orders));
19         String res = sB.waitForReceive().getLabel();
20         switch (res){
21             case "accept": // handle the order
22             case "reject": // terminate
23         }
24     }
25     else { // handled with the distributor
```

```
26          Public pD = cD.toPrivate(co2).tell(5 * 1000);
27          try {
28              Session sD = pD.waitForSession();
29              sD.send("req", getOutOfStockItems(orders));
30              try{
31                  switch (sD.waitForReceive().getLabel()) {
32                  case "no": sB.send("unavailable"); break;
33                  case "ok":
34                      sB.send("price", getPrice(orders));
35                      try{
36                          String res =
37                            sB.waitForReceive().getLabel();
38                          switch (res) {
39                          case "accept": // handle the order
40                          case "reject": // terminate
41                          }
42                      }
43                      catch (ContractViolationException e) {
44                          //the buyer is culpable, terminate
45                      }
46                  }
47              } catch (ContractViolationException e){
48                  //the distributor did not respect its contract
49                  sB.send("unavailable");
50              }
51          }
52          catch (ContractExpiredException e) {
53              //no distributor found
54              sB.send("unavailable");
55          }
56      }
57  } catch(ContractViolationException e){/*the buyer is culpable*/}
```

After advertising the contract `cB`, the store waits for a session `sB` with the buyer (lines 4–5). After the session is established, the store can receive `addtocart` multiple times: for each `addtocart`, it saves the corresponding item identifier in a list.

The loop terminates when the buyer selects `checkout`. If all requested items are available, the store sends the total `price` to the buyer (line 18). After that, the store expects either `accept` or `reject` from the buyer. If the buyer does not respect his deadlines, an exception is thrown, and it is caught at line 57. If the buyer replies on time, the store advertises the contract `cD`, and waits for a session `sD` with the distributor (lines 26–28). If the session is not established within 5 seconds, an exception is thrown. The store handles the exception at line 52, by sending `unavailable` to the buyer. If a session with the distributor is established within the deadline, the store requests the unavailable items, and waits for a response (line 31). If the distributor sends `no`, the store answers `unavailable` to the buyer (line 32). If the distributor sends `ok`, then the interaction between store and buyer proceeds as if the items were in stock. If the distributor does not reply within the deadline, an exception is thrown. The store handles it at line 47, by sending `unavailable` to the buyer. An untimed specification of this store is proved honest in [11]. We conjecture that also this timed version of the store respects contracts in all possible contexts.

Chapter 4

Decentralizing behavioural contracts on Bitcoin

In Chapter 3, we presented an architecture for contract-oriented computing based on a centralized middleware. The middleware allows mutually distrusting services with *compliant* contracts to safely interact, monitoring sessions and penalizing those who violated their contracts. We also validated our middleware in Section 3.4, showing that it enhances some security aspects in the development of distributed services, in particular to help programmers to reduce the effort of considering possible misbehaviours of their counterparts.

Under this setting, the middleware avoids the need for trusting on unknown third-party applications. However, implementing the middleware as a centralized entity makes the whole system vulnerable to adversaries who take control of the middleware itself, or manage to exclude it from the network.

The limitations of centralized approaches have been largely investigated in the research field of computer security, but recently cryptocurrencies and blockchain technologies like Bitcoin have pushed forward the concept of decentralization. A description of Bitcoin, its consensus mechanism and its applications is provided in Section 2.4, while a comparison to similar technologies is provided in Chapter 5. In particular, these sections illustrate as Bitcoin natively supports a limited model of contracts (usually called *smart* contracts), expressed in the form of scripts, and which can specify payments in their clauses.

On the other hand, Bitcoin allows external platforms to store metadata in its transactions, and several third-party services exploit this feature to store in the blockchain tamper-proof records produced by the execution of their advanced contracts. In this context, a sequence of *platform-specific* messages can be abstracted as a *subchain* inside the Bitcoin blockchain.

Except for the trivial case of contracts which admit any trace, in general there may exist *inconsistent* subchains which represent incorrect contract executions. A crucial issue is how to make it difficult, for an adversary, to subvert the execution of a contract by making its subchain inconsistent. Existing approaches either postulate that subchains are always consistent, or give weak guarantees about their security (for instance, they are susceptible to Sybil attacks [12]).

Thus, in this second part of our dissertation, we propose a consensus protocol, based on Proof-of-Stake, to incentivize platform nodes (called meta-nodes) to extend consistently the subchain. We also evaluate the security of our protocol, and, in Chapter 6 we show how to exploit it as the basis for extending the model of contracts supported by Bitcoin.

4.1 On subchains and consistency

We assume a set $\mathsf{A}, \mathsf{B}, \ldots$ of participants, who want to append messages $\mathtt{a}, \mathtt{b}, \ldots$ to the subchain. A *label* is a pair containing a participant A and a message $\mathtt{a}$, written $\mathsf{A} : \mathtt{a}$. *Subchains* are finite sequences of labels, written $\mathsf{A}_1 : \mathtt{a}_1 \cdots \mathsf{A}_n : \mathtt{a}_n$, which are embedded in the Bitcoin blockchain. The intuition is that A_1 has embedded the message $\mathtt{a}_1$ in some transaction $\mathtt{t}_1$ of the Bitcoin blockchain, then A_2 has appended some transaction $\mathtt{t}_2$ embedding $\mathtt{a}_2$, and so on. For a subchain η, we write $\eta\ \mathsf{A} : \mathtt{a}$ for the subchain obtained by appending $\mathsf{A} : \mathtt{a}$ to η.

In general, labels can also have side effects on the Bitcoin blockchain: we represent with $\mathsf{A} : \mathtt{a}(v \to \mathsf{B})$ a label which also transfers v₿ from A to B. When this message is on the subchain, it also acts as a standard currency transfer on the Bitcoin blockchain, which makes v₿ in a transaction of A redeemable by B. When the value v is zero or immaterial, we simply write $\mathtt{a}$ instead of $\mathtt{a}(v \to \mathsf{B})$.

A crucial insight is that not all possible sequences of labels are valid subchains:

to define the *consistent* ones, we interpret subchains as traces of *Labelled Transition Systems* (LTS). Formally, an LTS is a tuple $(Q, L, q_0, \to)$, where:

- Q is a set of states (ranged over by $q, q', \ldots$);
- L is a set of labels (in our case, of the form $\mathsf{A} : \mathtt{a}$);
- $q_0 \in Q$ is the initial state;
- $\to \subseteq Q \times L \times Q$ is a transition relation.

As usual, we write $q \xrightarrow{\mathsf{A}:\mathtt{a}} q'$ when $(q, \mathsf{A}:\mathtt{a}, q') \in \to$, and, given a subchain $\eta = \mathsf{A}_1 : \mathtt{a}_1 \cdots \mathsf{A}_n : \mathtt{a}_n$, we write $q \xrightarrow{\eta} q'$ whenever there exist $q_1, \ldots, q_n$ such that:

$$q \xrightarrow{\mathsf{A}_1:\mathtt{a}_1} q_1 \xrightarrow{\mathsf{A}_2:\mathtt{a}_2} \cdots \xrightarrow{\mathsf{A}_n:\mathtt{a}_n} q_n = q'$$

We require that the relation $\to$ is *deterministic*, i.e. if $q \xrightarrow{\mathsf{A}:\mathtt{a}} q'$ and $q \xrightarrow{\mathsf{A}:\mathtt{a}} q''$, then it must be $q' = q''$.

The intuition is that the subchain has a state (initially, q_0), and each message updates the state according to the transition relation. More precisely, if the subchain is in state q, then a message $\mathtt{a}$ sent by A makes the state evolve to q' whenever $q \xrightarrow{\mathsf{A}:\mathtt{a}} q'$ is a transition in the LTS.

Note that, for some state q and label $\mathsf{A} : \mathtt{a}$, it may happen that no state q' exists such that $q \xrightarrow{\mathsf{A}:\mathtt{a}} q'$. In this case, if q is the current state of the subchain, we want to make hard for a participant (possibly, an adversary trying to tamper with the subchain) to append such message. Informally, a subchain $\mathsf{A}_1 : \mathtt{a}_1 \cdots \mathsf{A}_n : \mathtt{a}_n$ is *consistent* if, starting from the initial state q_0, it is possible to find states $q_1, \ldots, q_n$ such that from each q_i there is a transition labelled $\mathsf{A}_{i+1} : \mathtt{a}_{i+1}$ to q_{i+1}.

Definition 4.1.1 (Subchain consistency)**.** We say that a subchain η is *consistent* whenever there exists q such that $q_0 \xrightarrow{\eta} q$.

Note that, if a subchain is consistent, then by determinism we have that the state q_n exists and is unique. In other words, a consistent sequence of messages uniquely identifies the state of the subchain.

Example 4.1.1. To illustrate consistency, consider a smart contract $\texttt{FACTORS}_n$ which rewards with 1฿ each participant who extends the subchain with a new prime

factor of n. The contract accepts two kinds of messages:

- $\mathtt{send}_p$, where p is a natural number;
- $\mathtt{pay}_p(1 \to \mathsf{A})$, meaning that A receives a reward for the factor p;

The states of the contract can be represented as sets of triples (A, p, b), where b is a boolean value indicating whether A has been rewarded for the factor p. The initial state is $\emptyset$. We define the transition relation of $\mathtt{FACTORS}_n$ as follows:

- $S \xrightarrow{\mathsf{A} : \mathtt{send}_p} S'$, iff p is a prime factor of n, $(\mathsf{B}, p, b) \notin S$ for any B and b, and $S' = S \cup \{(\mathsf{A}, p, 0)\}$;
- $S \xrightarrow{\mathsf{F} : \mathtt{pay}_p(1 \to \mathsf{A})} S'$, iff $(\mathsf{A}, p, 0) \in S$ and $S' = (S \setminus \{(\mathsf{A}, p, 0)\}) \cup \{(\mathsf{A}, p, 1)\}$.

Consider now the following subchains for $\mathtt{FACTORS}_{330}$, where F is the participant who issues the contract, and M is an adversary:

1. $\eta_1 = \mathsf{A} : \mathtt{send}_{11} \quad \mathsf{B} : \mathtt{send}_2 \quad \mathsf{F} : \mathtt{pay}_{11}(1 \to \mathsf{A}) \quad \mathsf{F} : \mathtt{pay}_2(1 \to \mathsf{B})$
2. $\eta_2 = \mathsf{A} : \mathtt{send}_{11} \quad \mathsf{F} : \mathtt{pay}_{11}(1 \to \mathsf{A}) \quad \mathsf{M} : \mathtt{send}_{11}$
3. $\eta_3 = \mathsf{M} : \mathtt{send}_{229} \; \mathsf{F} : \mathtt{pay}_{229}(1 \to \mathsf{M})$
4. $\eta_4 = \mathsf{A} : \mathtt{send}_{11} \quad \mathsf{F} : \mathtt{pay}_{11}(1 \to \mathsf{M})$

The subchain η_1 is consistent, because both A and B send new factors and get their rewards. The subchains η_2 and η_3 are inconsistent, because 11 sent by M is not fresh, and 229 is not a factor of 330. Finally, the subchain η_4 is inconsistent, because M gets the reward that should have gone to A. □

Similarly to Bitcoin, we do not aim at guaranteeing that a subchain is *always* consistent. Indeed, also in Bitcoin a miner could manage to append a block with invalid transactions: in this case, as discussed in Section 2.4, the Bitcoin blockchain forks, and the other miners must choose which branch to follow. However, honest miners will neglect the branch with invalid transactions, so eventually (since honest miners detain the majority of computational power), that branch will be abandoned by all miners.

For subchain consistency we adopt a similar notion: we assume that an adversary can append a label $\mathsf{A} : \mathtt{a}$ such that $q_n \not\xrightarrow{\mathsf{A}:\mathtt{a}}$, so making the subchain inconsistent. However, upon receiving such label, honest nodes will discard it. To formalise their behaviour, we define below a function Γ that, given a subchain η (possibly

inconsistent), filters all the invalid messages. Hence, $\Gamma(\eta)$ is a consistent subchain.

Definition 4.1.2 (Branch pruning)**.** We inductively define the endofunction Γ on subchains as follows, where ϵ denotes the empty subchain:

$$\Gamma(\epsilon) = \epsilon \qquad \Gamma(\eta\ \mathsf{A} : \mathtt{a}) = \begin{cases} \Gamma(\eta)\ \mathsf{A} : \mathtt{a} & \text{if } \exists q, q' : q_0 \xrightarrow{\Gamma(\eta)} q \xrightarrow{\mathsf{A}:\mathtt{a}} q' \\ \Gamma(\eta) & \text{otherwise} \end{cases}$$

In order to model which labels can be appended to the subchain without breaking its consistency, we introduce below the auxiliary relation $\models$. Informally, given a consistent subchain η, the relation $\eta \models \mathsf{A} : \mathtt{a}$ holds whenever the subchain $\eta\,\mathsf{A} : \mathtt{a}$ is still consistent.

Definition 4.1.3 (Consistent update)**.** We say that $\mathsf{A} : \mathtt{a}$ is a *consistent update* of a subchain η, denoted with $\eta \models \mathsf{A} : \mathtt{a}$, iff the subchain $\Gamma(\eta)\ \mathsf{A} : \mathtt{a}$ is consistent.

Example 4.1.2. Recall the subchain $\eta_2 = \mathsf{A} : \mathtt{send}_{11}\ \mathsf{F} : \mathtt{pay}_{11}(1 \to \mathsf{A})\ \mathsf{M} : \mathtt{send}_{11}$ from Example 4.1.1. We have that $\mathsf{B} : \mathtt{send}_2$ is a consistent update of η_2, because $\Gamma(\eta_2)\ \mathsf{B} : \mathtt{send}_2 = \mathsf{A} : \mathtt{send}_{11}\ \mathsf{F} : \mathtt{pay}_{11}(1 \to \mathsf{A})\ \mathsf{B} : \mathtt{send}_2$ is consistent. □

4.2 A protocol for consensus on Bitcoin subchains

Assume a network of mutually distrusted nodes $\mathsf{N}, \mathsf{N}', \ldots$, called *meta-nodes* to distinguish them from the nodes of the Bitcoin network. Meta-nodes receive messages from participants (also mutually distrusting) which want to extend the subchain. The goal is to allow honest participants (i.e., those who follow the protocol) to perform consistent updates of the subchain, while disincentivizing adversaries who attempt to make the subchain inconsistent.

To this purpose, this protocol is based on *Proof-of-Stake* (PoS) with the assumption that the overall stake retained by honest participants is greater than the stake of dishonest ones[1]. The stake is needed by meta-nodes, which have to vote for approving messages sent by participants. These messages are embedded into Bitcoin transactions, and called *update requests*. Let $\mathsf{UR}[\mathsf{A} : \mathsf{a}]$ denote the update request issued by A to append the message a to the subchain. In order to vote an update request, a meta-node must invest κ฿ on it, where κ is a constant specified by the protocol. An update request needs the vote of a single meta-node. The protocol requires meta-nodes to vote a request $\mathsf{UR}[\mathsf{A} : \mathsf{a}]$ only if $\mathsf{A} : \mathsf{a}$ is a consistent update of the current subchain η, i.e. if $\eta \models \mathsf{A} : \mathsf{a}$[2]. To incentivize meta-nodes to vote their update requests, participants pay them a constant *fee*, which can be redeemed by meta-nodes when the update request is appended to the subchain.

[1] Note that a similar hypothesis, but related to computational power rather than stake, holds in Bitcoin, where honest miners are supposed to control more computational power than dishonest ones.

[2] We assume that all meta-nodes agree on the Bitcoin blockchain; since η is a projection of the blockchain, they also agree on η.

1. Upon receiving an update request $\mathsf{UR}[\mathsf{A} : \mathsf{a}]$, a meta-node checks its consistency, $\eta \models \mathsf{A} : \mathsf{a}$. If so, it votes the request, and adds it to the request pool;
2. when Δ expires, the arbiter signs all the well-formed URs in the request pool;
3. all requests signed by the arbiter are sent to the Bitcoin miners, to be published on the blockchain. The first to be mined, indicated with UR_i, is the i-th label of the subchain.

Figure 4.1: Summary of a protocol stage i.

The protocol is defined in Figure 4.1. It is organised in *stages*. The protocol ensures that *exactly one* label $\mathsf{A} : \mathtt{a}$ is appended to the subchain for each stage i. This is implemented by appending a corresponding transaction $\mathsf{UR}_i[\mathsf{A} : \mathtt{a}]$ to the Bitcoin blockchain. To guarantee its uniqueness, the protocol exploits an *arbiter* T, namely a distinguished node of the network which is assumed honest (this hypothesis is discussed in Section 4.3). The main steps of the protocol are now described.

At step 1 of the stage i of the protocol, a meta-node (say, N) votes an update request (as detailed in Section 4.5).

In order to do this, N must confirm some of the past C updates (where $C \geq 1$ is the *cutoff window*, a constant fixed by the protocol and described in Section 4.2.1). To confirm an update, N uses the $\kappa\text{฿}$ to pay the meta-nodes who respectively appended each chosen update UR_j (with $i - C \leq j < i$) to the subchain. The way to choose the updates UR_j to be confirmed is called *refund policy* and is deepened in Section 4.2.1. After voting, N adds $\mathsf{UR}[\mathsf{A} : \mathtt{a}]$ to the *request pool*, i.e. the set of all voted requests of the current stage (emptied at the beginning of each stage). This voting step has a fixed duration Δ, specified by the protocol (the choice of Δ is discussed in Chapter 6).

At step 2, which starts when Δ expires, the arbiter T signs all *well-formed* request transactions, i.e., those respecting the format defined in Section 4.5.

At step 3, meta-nodes send the requests signed by T to the Bitcoin network. The mechanism described in Section 4.5 ensures that, at each stage i, exactly one transaction, denoted $\mathsf{UR}_i[\mathsf{A} : \mathtt{a}]$, is put on the Bitcoin blockchain. When this happens, the label $\mathsf{A} : \mathtt{a}$ is appended to the subchain.

Summarizing, the protocol depends on the parameters $\Pi = (C, \kappa, f, r)$, which are, respectively, the cutoff window size, the amount required to vote an update request, the fee payed by the client, and the maximum transferable amount in special updates in the form $\mathsf{A} : \mathtt{a}(v \to \mathsf{B})$ (where, by definition, $v \leq r$).

4.2.1 Refund policies

A refund policy can be formally defined as a function Θ that, given a subchain η and the protocol parameters $\Pi = (C, \kappa, f, r)$, outputs a sequence of refunds $\rho^i = (\rho^i_1 \dots \rho^i_C)$, where:

- ρ^i_j represents, at each stage i of the protocol, the amount to pay to the meta-nodes who voted UR_{i-j}, for every j s.t. $1 \leq j \leq C$ (only updates inside the current cutoff window can be refunded);
- $\sum_{j=1}^{C} \rho^i_j = \kappa + f$ (the policy specifies how to split the vote and the fee among the voters of the updates inside the cutoff window).

To enforce good behaviour, updates whose voters did not follow the prescribed policy are considered not *refundable*. This means honest meta-nodes penalize not only inconsistent labels, but also illegal refunds.

Definition 4.2.1 (Refundable update)**.** Let $\eta^{[1\dots k]}$ be the subchain after the completion of the k-th protocol stage, let $\mathsf{UR}_j[\mathsf{A} : \mathsf{a}]$ be a published update, and $\bar{\rho}^j$ the refund made by its voter. Then, we say UR_j is *refundable* if and only if it is consistent ($\eta^{[1\dots(j-1)]} \models \mathsf{A} : \mathsf{a}$) and it follows the refund policy ($\bar{\rho}^j = \Theta(\eta^{[1\dots(j-1)]}, \Pi)$).

Thus, at each stage i, we can define the set of indexes of refundable updates in the current cutoff window. Suppose that the subchain starts with C predetermined and consistent updates $\mathsf{UR}_{-i}, 1 \leq i \leq C$, then:

$$\xi^0 = \{1 \leq j \leq C\} \tag{4.1}$$

since all initial updates are considered refundable. Then, for $i > 0$:

$$\xi^i = \{1 \leq j \leq C : \mathsf{UR}_{i-j} \text{ is refundable according to } \Theta\} \tag{4.2}$$

Note that checking if the voter of the j-th update (with $j < i$) has followed the refund policy just requires to examine the updates with index $k \leq j$. Thus, this check may depend only on ξ^k and never on ξ^i.

As an example, some possible refund policies are now presented.

newest-first This policy refunds only the newest refundable update in the cutoff

window (if any), the newest in general otherwise:

$$\rho_j^i = \begin{cases} \kappa + f & \text{if } \xi^i \neq \emptyset \wedge j = min(\xi^i) \\ \kappa + f & \text{if } \xi^i = \emptyset \wedge j = 1 \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

oldest-first This policy refunds the oldest consistent update (if any), the oldest in general otherwise (note that it coincides with the `newest-first` if $C = 1$):

$$\rho_j^i = \begin{cases} \kappa + f & \text{if } \xi^i \neq \emptyset \wedge j = max(\xi^i) \\ \kappa + f & \text{if } \xi^i = \emptyset \wedge j = C \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

4.2.2 Proof-of-Burn

To expand the possibilities for meta-nodes and increase the security of the protocol, as will be deepened in Section 4.3, the sequence of refunds ρ can be extended to include a special value ρ_0. This value represents the amount that should be paid to a pre-set fictional address (e.g. an *all-zero* address). Refunding such an address effectively corresponds to burning the money sent, making it unspendable.

With this enhancement, the policies defined previously can be improved, removing the case $\xi^i = \emptyset$ and adding:

$$\rho_0^i = \begin{cases} \kappa + f & \text{if } \xi^i = \emptyset \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

which can be interpreted as follows: if no update in the cutoff window is refundable, burn vote and fee. The variants of the previous policies, after the inclusion of the Proof-of-Burn condition, are called `newest-first-pburn` and `oldest-first-pburn`. This change avoids the (forced) confirmation of a non-refundable update, that is allowed in the previous definitions of the `newest-first` and `oldest-first` policies.

Now, recall that in Section 4.2 the condition $C \geq 1$ is provided. However, the choice $C = 1$ makes sense only if there is the possibility of burning the vote. Vice versa, voters would have no other choice besides confirming the previous update (refundable or not). Introducing the Proof-of-Burn, instead, the following policy for $C = 1$ can be defined and used.

harsh policy This policy refunds the previous update if refundable, burns the money otherwise:

$$\rho_1^i = \begin{cases} \kappa + f & \text{if } \xi^i = \{1\} \\ 0 & \text{otherwise} \end{cases} \qquad \rho_0^i = \begin{cases} \kappa + f & \text{if } \xi^i = \emptyset \\ 0 & \text{otherwise} \end{cases} \tag{4.6}$$

4.3 Basic properties of the protocol

We now establish some basic properties of our protocol. Hereafter, we assume that honest nodes control the majority of the total stake of the network, denoted by S. Further, we assume that the overall stake required to vote pending update requests is greater than the overall stake of honest meta-nodes.

4.3.1 Adversary power

An honest meta-node votes as many requests as is allowed by its stake. Hence, if its stake is h, it votes h/κ requests per stage. Consequently, the rest of the network — which may include dishonest meta-nodes not following the protocol — can vote at most $(S-h)/\kappa$ requests[3].

Then:

Lemma 4.3.1. *The probability that an honest meta-node with stake h updates the subchain is at least h/S at each stage.*

Since we assume that honest meta-nodes control the majority of the stake, Lemma 4.3.1 also limits the capabilities of the adversary:

Lemma 4.3.2. *If the global stake of honest meta-nodes is S_H, then dishonest ones update the subchain with probability at most $(S-S_H)/S$ at each stage.*

Although inconsistent updates are ignored by honest meta-nodes, their side effects as standard Bitcoin transactions (i.e. trasfers of v₿ from A to B in labels $\mathsf{A} : \mathsf{a}(v \to \mathsf{B})$) cannot be revoked once they are included in the Bitcoin blockchain.

Even though the goal of the protocol is to let meta-nodes get revenues proportionate to their probability of updating the subchain (as defined in Lemma 4.3.1 and Lemma 4.3.2), the adversary might exploit these side effects to earn more then she owes by publishing inconsistent updates. Therefore, we show how the incentive system in our protocol reduces the feasibility of such inconsistent updates.

[3]Note that assuming the ability of the adversary to delay some messages, thus reducing the honest meta-nodes actual voting power (since their voted requests might not reach the request pool), is equivalent to consider an adversary with a higher stake.

According to Lemma 4.3.2, if M has stake m, and the other meta-nodes are honest, then M has probability at most m/S of extending the subchain in a given stage of the protocol. Since stages can be seen as independent events, we obtain the following:

Lemma 4.3.3. *The probability that an adversary with stake m saturates a cutoff window with her updates only (consistent or not) is μ^C, where C is the cutoff window size, and $\mu = m/S$.*

To simplify the terminology, hereafter we consider a consistent update to be also refundable[4].

Now, assume M manages to publish C consecutive updates (consistent or inconsistent) starting from index j, with probability given by Lemma 4.3.3. M can use each update at index $j < k \leq j + C$ to recover her vote κ and eventually the fee f for her previous update at index $k - 1$, such that only the last update at index $j + C$ remains unrefunded.

In particular, if the protocol specifies a refund policy which not admit the *Proof-of-Burn* described in Section 4.2.2, at least a honest update at index $i > j + C$ has to necessarily refund M of $(\kappa + f)$฿, since she saturated the cutoff window with her updates only. Consequently, following this strategy the attacker does not lose any deposit and possibly earns an additional extra revenue r for each inconsistent update she published, if any. This extra revenue r models the case where M induces a victim A to publish an inconsistent update in the form $\mathsf{A} : \mathsf{a}(r \rightarrow \mathsf{M})$.

Also note that, if M cannot manage to saturate the cutoff window immediately, she can delay the completion of the attack by publishing at least one inconsistent update every C ones on the subchain (to keep refunding herself the vote and the fee). We call the above behaviour of M (and all its variants) the *self-compensation attack*.

Finally, observe that the choice of the protocol parameters and, particularly, the refund policy is crucial to force the honest strategy to be more profitable than

[4]Publishing a consistent update that is not refundable does not break the consistency of the subchain, but it causes the meta-node who voted the update to be (eventually) not refunded for its effort. Therefore, this behaviour cannot be considered an attack.

any dishonest one. To support this claim, in what follows we show a dishonest strategy which exploits a variant of the attack, called the *reversed self-compensation attack*, and we prove that it is always more profitable than the honest strategy whether the chosen refund policy is `newest-first`.

4.3.2 Reversed self-compensation attack

Assume an adversary M that manages to append two updates on the subchain, the first with index i, and the second with index $i+1 < j \leq i+C$. Suppose that the update at index i is consistent, then the honest meta-node that publishes the update at index $i+1$ (recall, the considered refund policy is `newest-first`) refunds $(\kappa + f)$₿ to M. Now M can use again these funds to publish a new inconsistent update at index j, refunding again her update at index i (thus also violating the refund policy). So M manages to earn the undeserved extra revenue r without having lost neither κ nor f, therefore performing a special case of the self-compensation attack.

Now, consider a conservative strategy where the attacker M at first tries to publish consistent updates only. When she manages to do so for a while, she tries to publish just one inconsistent update until the last consistent update published is beyond the cutoff window, then reverses again to consistent updates.

Let μ be the probability M has in successfully publishing an update, and suppose she published the last update, consistent. We show that the expected payoff ϕ_D of M, when she follows the described dishonest strategy, is *always* greater than the expected payoff ϕ_H she can get if she follows the honest strategy. The analysis is limited to the subsequent C updates, since the two strategies coincide afterwards, and holds only for $C \geq 2$ (with $C = 1$, meta-nodes have no choice from refunding the last update, so an inconsistent update is always more profitable than a consistent one). From the hypothesis, it follows that:

$$\phi_D = \mu f + \sum_{i=1}^{C-1} \mu(1-\mu)^{i-1}(f + r + \mu f(C-1-i)) \tag{4.7}$$

$$\phi_H = \mu f C \tag{4.8}$$

In Eq. 4.7, μf describes the payoff of M for the first update she publishes, while the rest denote the sum of the revenues obtained by publishing one or more update in the subsequent cutoff window, weighted for the respective probabilities to occur. Then, the gain M obtains by following the dishonest strategy is:

$$\begin{aligned}\phi_D - \phi_H &= \mu f(1-C) + \mu \sum_{i=1}^{C-1}(1-\mu)^{i-1}(f + r + \mu f(C-1-i)) \\ &= \mu f \left(\sum_{i=1}^{C-1}(1-\mu)^{i-1}(1+\mu(C-1-i)) - (C-1) \right) + \mu r \sum_{i=1}^{C-1}(1-\mu)^{i-1} \\ &= \mu r \sum_{i=1}^{C-1}(1-\mu)^{i-1} \end{aligned} \tag{4.9}$$

The result of eq. (4.9) is justified by the following Lemma 4.3.4. Since $0 < \mu < 1$, $\forall r$ s.t. $r > 0$ we get $\phi_D > \phi_H$. This means that, indipendently from the chosen protocol parameters Π, a protocol that uses the refund policy `newest-first` admits at least one dishonest strategy which is always more profitable than the honest one. Note also that a similar result can be obtained for the `oldest-first` policy.

Lemma 4.3.4. *For $C \geq 2$, it holds:*

$$\sum_{i=1}^{C-1}(1-\mu)^{i-1}(1+\mu(C-1-i)) - (C-1) = 0 \tag{4.10}$$

Proof.

$$
\begin{aligned}
&\sum_{i=1}^{C-1}(1-\mu)^{i-1}(1+\mu(C-1-i)) - (C-1) \\
&= \sum_{j=0}^{C-2}\left((-\mu)^j \sum_{i=j}^{C-2}\binom{i}{j} - (-\mu)^{j+1}\sum_{i=j}^{C-2}\binom{i}{j}(C-2-i)\right) - (C-1) \\
&= (-\mu)^0 \sum_{i=0}^{C-2}\binom{i}{0} - (C-1) + \sum_{j=1}^{C-2}(-\mu)^j \sum_{i=j}^{C-2}\binom{i}{j} - \sum_{j=1}^{C-1}(-\mu)^j \sum_{i=j-1}^{C-2}\binom{i}{j-1}(C-2-i) \\
&= \sum_{j=1}^{C-2}(-\mu)^j\left(\sum_{i=j}^{C-2}\binom{i}{j} - \sum_{i=j-1}^{C-3}\binom{i}{j-1}(C-2-i)\right) \\
&= \sum_{j=1}^{C-2}(-\mu)^j \sum_{i=j}^{C-2}\left(\binom{i}{j} - \binom{i-1}{j-1}(C-1-i)\right) \qquad (4.11)
\end{aligned}
$$

From the following Lemma 4.3.5, it follows that the coefficients of the polynomial in Equation (4.11) are all zero, thus proving the above Lemma 4.3.4. □

Lemma 4.3.5. *For $n \geq 1$, it holds:*

$$\sum_{i=j}^{n}\left(\binom{i}{j} - \binom{i-1}{j-1}(n+1-i)\right) = 0 \qquad 1 \leq j \leq n \qquad (4.12)$$

Proof. We prove it by induction over n. The base case is $n = 1$, therefore $j = 1$.

$$\sum_{i=1}^{1}\left(\binom{i}{1} - \binom{i-1}{0}(2-i)\right) = 1 - 1 = 0 \qquad (4.13)$$

For the inductive step:

$$\sum_{i=j}^{n+1}\left(\binom{i}{j} - \binom{i-1}{j-1}(n+2-i)\right)$$

$$= \sum_{i=j}^{n} \left(\binom{i}{j} - \binom{i-1}{j-1}(n+1-i) \right) + \binom{n+1}{j} - \sum_{i=j}^{n+1} \binom{i-1}{j-1}$$
$$= \binom{n+1}{j} - \sum_{i=j}^{n+1} \binom{i-1}{j-1}$$

To conclude, the results of the following Lemma 4.3.6 are needed. □

Lemma 4.3.6. *For $n \geq 1$ it holds:*

$$\binom{n}{k} = \sum_{i=k}^{n} \binom{i-1}{k-1} \qquad 1 \leq k \leq n \tag{4.14}$$

Proof. The proof is again by induction over n.The base case is $n = 1$, thus $k = 1$.

$$\binom{1}{1} = 1 = \sum_{i=1}^{1} \binom{0}{0} \tag{4.15}$$

For the inductive step:

$$\binom{n+1}{k} = \binom{n}{k} + \binom{n}{k-1} \qquad 1 \leq k \leq n \tag{4.16}$$

$$= \binom{n}{k-1} + \sum_{i=k}^{n} \binom{i-1}{k-1}$$
$$= \sum_{i=k}^{n+1} \binom{i-1}{k-1}$$

□

4.3.3 Trustworthiness of the arbiter

The protocol uses an arbiter T to ensures that only one transaction per stage is appended to the blockchain, and that its choice is random as well. In order to simplify the description of the protocol, the arbiter T has been assumed to behave honestly. In fact, the arbiter introduces a point of centralization but it can be observed that it does not play the role of a trusted authority. As an example, im-

plementing the middleware descripted in Chapter 3 as a centralized entity (e.g., a private server) implies that the services in the network cannot ensure the fulfillment of the contract rules by their counterparties, as this task is performed by the middleware, and services must trust on it.

Vice versa, in the decentralized settings of the protocol, the update requests to be voted are chosen by the meta-nodes, and once they are added to the request pool, the arbiter is expected to sign all of them, without taking part on the validation nor in the voting. Recall that transactions are first signed by clients and voters, thus neither the arbiter nor other nodes can modify their content once sent to the request pool. This means that the arbiter can only refuse to add its sign to a subset of them, i.e. prevent their publication (which corresponds to isolate some meta-nodes in the network). But since everyone can inspect the request pool, any misbehaviour of the arbiter can be detected by the meta-nodes, that can proceed to replace it. Therefore, the presence of the arbiter cannot affect the decentralization of the approach but, without the assumption of its honesty, it would be necessary to take in consideration some additional malicious behaviours like, e.g., temporary *Denial-of-Service* attacks performed by the arbiter. This is still an open issue: in particular, in Section 6.1 we discuss our future research directions on algorithms that allows meta-nodes to safely replace the arbiter when a misbehaviour is detected.

4.4 Evaluation of the protocol

In this section we evaluate the security of the protocol, providing some analytical results. In particular, we illustrate a realistic attack scenario, and investigate how the choice of protocol parameters and the refund policy can disincentivize adversaries to behave dishonestly. We also examine how possible attacks to Bitcoin may affect subchains built on top of its blockchain.

4.4.1 Adversary strategy

To analyse possible attacks, consider an adversary who can craft any update (consistent or not), and controls one meta-node M with stake ratio $\mu = m/S$, where

$\mu \in [0; 1]$, m is the stake controlled by the adversary and S is the total stake of the network[5]. Suppose that each meta-node can vote as many update requests as possible, spending all its stake, and that the network is always saturated with pending updates, which globally amount to the entire stake of honest meta-nodes[6].

To evaluate its security, we model the protocol as a game, in which the attacker M is a player that adopts a possibly dishonest strategy, thus trying to publish either consistent or inconsistent updates. Conversely, the other players are the honest meta-nodes, that follow the protocol and therefore adopt a honest strategy, trying to publish consistent updates only. Suppose also that M follows an *optimal* strategy, i.e. according to the current state, the choice of voting a consistent or inconsistent update — at each protocol stage — is made with the goal of maximizing her final revenue. In particolar, the current state depends on the content of the current cutoff window, and not on the full history of the subchain:

Lemma 4.4.1. *The revenue of an update published by an adversary* M*, at the protocol stage i, depends only on the state of the cutoff window in that stage (i.e., $\eta^{[(i-C)\ldots(i-1)]}$), on the protocol parameters Π, on the refund policy Θ and on the adversary ratio μ.*

To justify Lemma 4.4.1, observe that, by definition of refund policy Θ, no update with index $j < i - C$ can be refunded, neither of the vote κ nor the fee f, in a protocol stage with index i. Thus, no matter what M chooses at the current stage, there is no additional revenue (but also no loss) for any update outside the cutoff window.

Now, let G be a function that maps a subchain $\eta^{[1\ldots k]}$ into a sequence of labels $s = \sigma_1 :: \ldots :: \sigma_k$. A label σ_i can assume one of the following values: `Inc`, which indicates an inconsistent update published by M; `Con`, which represents a consistent update published by M, and `Ext`, that denotes a (consistent) external update published by the rest of the network, assumed to be honest.

Also, let $s_C^k = G(\eta^{[(k-C)\ldots(k-1)]})$ be the sequence that represents the cutoff window

[5] Assuming a single adversary is not less general than having many non-colluding meta-nodes which carry on individual attacks. Indeed, in this setting meta-nodes do not join their funds to increase the stake ratio μ.

[6] Note that saying the update queue is not always saturated is equivalent to model an adversary with a stronger μ: this because honest meta-nodes cannot spend all their stake in a single protocol stage, i.e. reducing their actual *power*. Thus, studying this particular case will not give any additional contribution to the analysis.

state at the stage k This sequence is used to generate two new sequences $s^k_{\texttt{Con}} = s^k_C :: \texttt{Con}$ and $s^k_{\texttt{Inc}} = s^k_C :: \texttt{Inc}$ that represent the possible continuations of the chain if the adversary manages to publish the next update.

We also need a function ϕ that, given the protocol parameters Π and the refund policy Θ, takes a sequence s as input and computes the *a posteriori* attacker revenue associated to s. Moreover, let ϕ' be a variant of ϕ that, in addition, takes into account the possible refunds generated appending C `Ext` updates at the end of s (this models the case of an attack that terminates).

Through ϕ and ϕ', and given the attacker ratio μ, it is possible to define a payoff function Φ_d with depth d. Let $s = \sigma_1 :: \ldots :: \sigma_N$ be a sequence of length N, and $s' = \sigma_2 :: \ldots :: \sigma_N$ be the same sequence of s, where the first label is elided. The payoff function is defined recursively as:

$$\Phi_d(s) = \begin{cases} \phi(s) + (1-\mu)\Phi_{d-1}(s' :: \texttt{Ext}) + \mu \max(\Phi_{d-1}(s' :: \texttt{Con}), \Phi_{d-1}(s' :: \texttt{Inc})) & d > 1 \\ \phi'(s) & d = 1 \end{cases}$$

With all these ingredients, we can formulate the following:

Definition 4.4.1 (Optimal choice)**.** Given a sequence s^k_C that represents the state of the current cutoff window, the *optimal choice with depth* d for the adversary is to try to publish a consistent update if $\Phi_d(s^k_{\texttt{Con}}) > \Phi_d(s^k_{\texttt{Inc}})$, an inconsistent one otherwise.

In particular, consider an adversary that plans to meddle with the protocol for a limited amount of time, say for d stages, starting at stage n. In this scenario, the *optimal strategy* for the adversary would require, at each stage $n \leq k \leq n + d - 1$, to take the optimal choice with depth $n + d - k$.

This guarantees, to M, to get the maximum possible revenue at the end of the attack: in fact, the optimal choice of depth d (i.e., the initial choice) takes into account every possible evolution of the protocol for the duration of the attack, weighted for its probability to occur, and considers the best outcome at every step. At last, note that the optimal strategy cannot be well defined for an adversary that plans to attack the protocol for an indefinite number of stages; however,

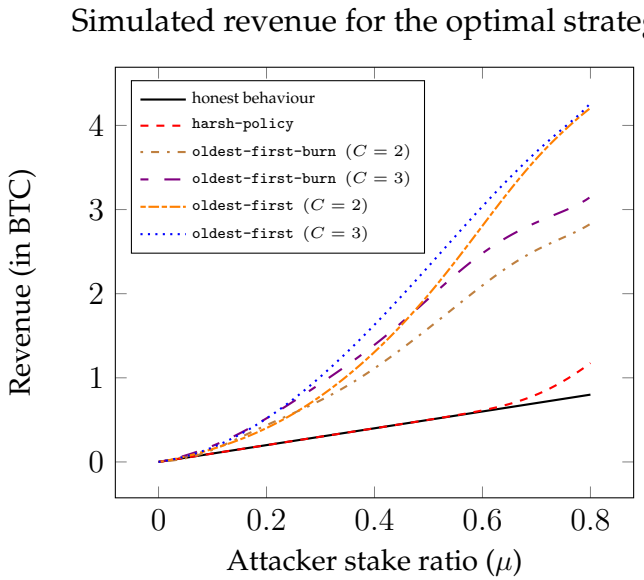


Figure 4.2: We simulated the revenue of an adversary M which participates in subchains of length 100 and uses the optimal strategy. Each curve represents the revenue of M as μ increases, and for a different refund policy. We fixed the protocol parameter as follows: $f = 0.01$, $r = 0.03$ and $v = 0.1$, all conventionally expressed in bitcoins.

it can be effectively approximated by considering — at each step — to take the optimal choice of *fixed* depth d, provided d great enough.

4.4.2 Analytical results

In what follows, we show the results of the security analysis of the protocol, under the attack scenario in which the adversary adopts an optimal strategy, and only for the `harsh-policy`, which provides the best security performance among the policies shown in Section 4.2.1 (according to some simulated preliminary experiments shown in Figure 4.2). The results are summarized by the following:

Theorem 4.4.1. *If the protocol prescribes to use the `harsh-policy`, an adversary with stake ratio μ, that adopts an optimal strategy in pursuing an attack of arbitrary length, behaves necessarily honest if and only if $\mu < 1 - r/(\kappa + f)$.*

Proof. Note that, if the prescribed policy is the `harsh-policy`, any update after a `Con` must necessarily refund the vote and the fee to the adversary, independently

from its type. Thus, this additional gain can be included in the revenue, having:

- $\phi(-, \texttt{Con}) = f$

where the symbol '$-$' indicates an indifference condition.

On the other side, the revenue for an inconsistent update can be quantified in:

- $\phi(\texttt{Ext}, \texttt{Inc}) = r - \kappa$
- $\phi(\texttt{Inc}, \texttt{Inc}) = r + f$ (vote and fee are self-refunded)
- $\phi(\texttt{Con}, \texttt{Inc}) = r - \kappa$ (the refund of f and κ has already been counted for `Con`)

Note that, when considering to publish an inconsistent update, a cutoff that contains `Con` is equivalent to one with `Ext`. Finally, observe that computing the revenue in this way gives $\phi(-, \texttt{Ext}) = 0$, and therefore $\phi = \phi'$.

Now note that, at the terminal stages of the attack, the adversary is encouraged to publish consistent updates, since she has to publish at least two consecutive `Inc` to outdo the honest behaviour, because $\phi'(\texttt{Ext}, \texttt{Inc}) < \phi'(\texttt{Ext}, \texttt{Con})$, and the chances to do so decreases[7]. On the opposite, at the early stages, a high enough μ ensures that — on average — a sufficient number of consecutive `Inc` will be published to gain an advantage over the honest behaviour. And, in this case, since $\phi(\texttt{Inc}, \texttt{Inc}) \geq \phi(-, \texttt{Inc})$, it follows that if publishing an inconsistent update is the optimal choice at the step k, then it has been the optimal choice at the step $k - 1$ too.

Under these assumptions, we can conclude that the optimal strategy can be either completely honest or starting dishonest (always trying to publish `Inc`s) and then switching to the honest one as the end of the attack approaches. Thus, for our analysis, we can only consider the early stages, in which publishing inconsistent updates can be more profitable than publishing consistent ones. Here, the adversary that publishes `Inc` gains $\phi(\texttt{Ext}, \texttt{Inc})$ with probability $(1 - \mu)$, and $\phi(\texttt{Inc}, \texttt{Inc})$ with probability μ (recall that μ is the probability that the adversary manages to publish an update in a given stage). Therefore, the adversary chooses the honest

[7] Only exceptions are the rare cases in which a streak of consecutive `Inc` occurs in the terminal stages: here, the adversary is motivated to continue publishing inconsistent updates. But as soon as an attempt fails (an `Ext` breaks the sequence), and if the end is near enough, the adversary switches to the honest behaviour.

strategy right from the start if and only if:

$$\begin{aligned}
&\phi(-, \texttt{Con}) > (1-\mu)\cdot\phi(\texttt{Ext}, \texttt{Inc}) + \mu\cdot\phi(\texttt{Inc}, \texttt{Inc}) \\
&\iff f > r - \kappa + (f+\kappa)\mu \\
&\iff f > r - \kappa + (f+\kappa)\mu \\
&\iff \mu < \frac{f+\kappa-r}{f+\kappa} \\
&\iff \mu < 1 - \frac{r}{f+\kappa}
\end{aligned} \tag{4.17}$$

□

The value of f is assumed to be strictly smaller than κ, in order to incentivize the participation of meta-nodes to the protocol. In fact, with f very close to (or even greater than) κ, clients have no evident benefit from delegating meta-nodes to vote their updates (since the required economical effort does not change significantly). This is similar to provide a protocol with no fees, which is less attractive for meta-nodes to participate in. However, a large participation to the protocol reduces the possibility that single or colluding entities control the majority of the stake.

Under this assumption, the eq. (4.17) shows that the security of the protocol is essentially proportional to the ratio κ/r: the higher this ratio, the more convenient an honest behaviour becomes. Figure 4.3 finally illustrates how the *switch point* (i.e., the $min(\mu)$ such that $\phi_{\texttt{Inc}} \geq \phi_{\texttt{Con}}$) varies for different combinations of κ and r, with a small fixed fee ($f = 0.01$฿).

4.4.3 Security of the underlying Bitcoin blockchain

So far we have only considered direct attacks to our protocol, assuming the underlying Bitcoin blockchain to be secure. However, although Bitcoin has been secure in practice till now, some works have spotted some potential vulnerabilities of its protocol. These vulnerabilities could be exploited to execute *Sybil attacks* [12] and *selfish-mining attacks* [52], which might also affect subchains built on top of the Bitcoin blockchain.

In Sybil attacks on Bitcoin, honest nodes are induced to believe that the network

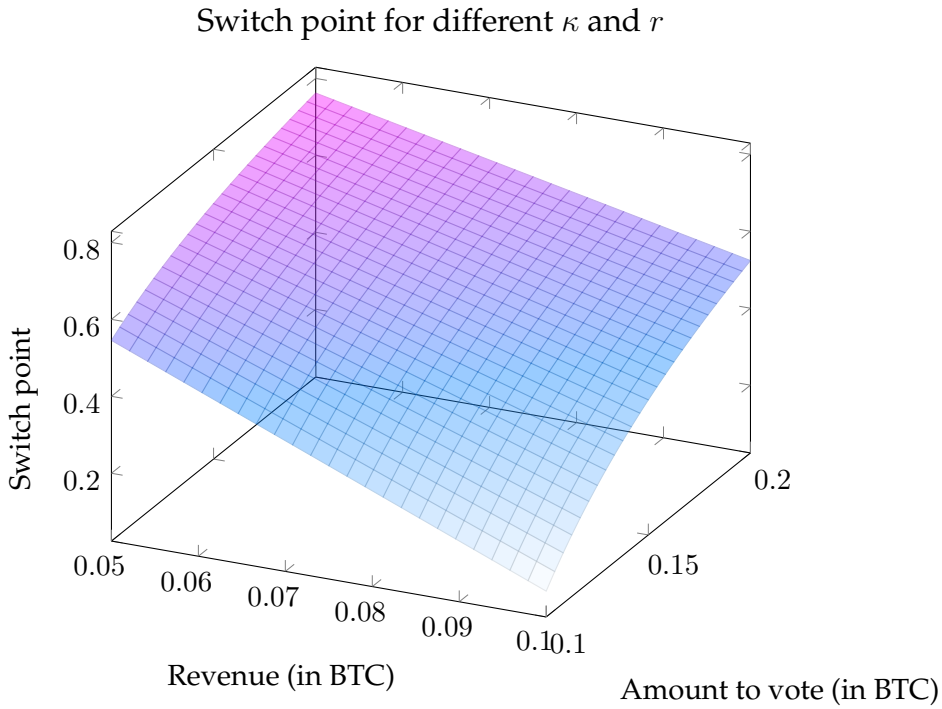


Figure 4.3: The plot shows how the switch point (the $min(\mu)$ such that $\phi_{\texttt{Inc}} \geq \phi_{\texttt{Con}}$) varies for different combinations of κ and r, when the refund policy is `harsh-policy` and f is fixed to 0.01₿.

is populated by many distinct participants, which instead are controlled by a single malicious entity. This attack is usually exploited to quickly propagate malicious information on the network, and to disguise honest participants in a consensus/reputation protocol, e.g. by overwhelming the network with votes of the adversary. In the selfish-mining attack [52], small groups of colluding miners manage to obtain a revenue larger than the one of honest miners. More specifically, when a selfish-mining pool finds a new block, it keeps it hidden to the rest of the network. In this way, selfish miners gain an advantage over honest ones in mining the next block. This is equivalent to keep a private fork of the blockchain, which is only known to the selfish-mining pool. Note that honest miners still mine on the public branch of the blockchain, and their hash rate is greater than selfish miners' one. Since, in the presence of a fork, the Bitcoin protocol requires to keep mining on the longest chain, selfish miners reveal their private fork to the network just before being overcome by the honest miners. Eyal and Sirer in [52] show that, under certain assumptions, this strategy gives better revenues than

honest mining: in the worst scenario (for the adversary), the attack succeeds if the selfish-mining pool controls at least $1/3$ of the total hashing power. Rational miners are thus incentivized to join the selfish-mining pool. Once the pool manages to control the majority of the hashing power, the system loses its decentralized nature. Garay, Kiayias and Leonardos in [53] essentially confirm these results: considering a core Bitcoin protocol, they prove that if the hashing power γ of honest miners exceeds the hashing power β of the adversary pool by a factor λ, then the ratio of adversary blocks in the blockchain is bounded by $1/\lambda$ (which is strictly greater than β). Thus, as β (the adversary pool size) approaches $1/2$, they control the blockchain.

Although these attacks are mainly related to Bitcoin revenues, they can affect the consistency of any subchain built on top of its blockchain. In particular, suitably adapted versions of these attacks allow adversaries to cheat meta-nodes about the current subchain state, forcing them to synchronize their local copy of the Bitcoin blockchain with invalid forks that will be discarded by the network in the future. In order to be protected against such attacks, meta-nodes should consider only *l-confirmed* transactions. Namely, if the last published blockchain block is B_n, they consider only those transactions appearing in blocks B_j with $j \leq n - l$. This means that an attacker would have to mine at least l blocks to force the revocation of a *l-confirmed* transaction. Rosenfeld [78] shows that, if an attacker controls 10% at most of the network hashing power, $l = 6$ is sufficient for reducing the risk of revoking a transaction to less than 0.1%.

4.5 Implementation in Bitcoin

In this section we show how our protocol can be implemented in Bitcoin. A label $\mathsf{A} : \mathtt{a}(v \to \mathsf{B})$ at position i of the subchain is implemented as the Bitcoin transaction $\mathsf{UR}_i[\mathsf{A} : \mathtt{a}(v \to \mathsf{B})]$ in Figure 4.4a, with the following outputs:

- the output of index 0 embeds the label $\mathsf{A} : \mathtt{a}$. This is implemented through an unspendable `OP_RETURN` script [24][8].
- the output of index 1 links the transaction to the previous element of the

[8] The `OP_RETURN` instruction allows to save 80 bytes metadata in a transaction; an out-script containing `OP_RETURN` always evaluates to false, hence it is unspendable.

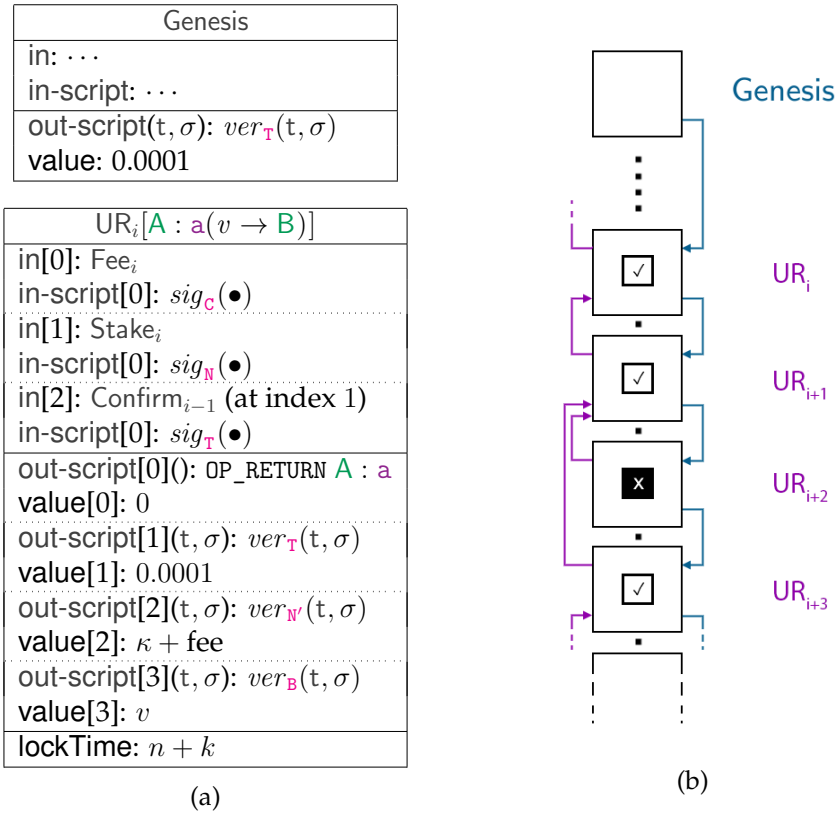


Figure 4.4: In (a), format of Bitcoin transactions used to implement our protocol. In (b), a subchain mantained through our protocol. Since UR_{i+2} contains an inconsistent update, the meta-node which voted it is not rewarded.

subchain, pointed by in[2]. This link requires the arbiter signature. Note that, since all the update requests in the same stage redeem the same output, exactly one of them can be mined.

- the output of index 2 implements the incentive mechanism. The script rewards the meta-node N′ which has voted a preceding UR_j in the subchain. Meta-node N′ can redeem from this output κ฿ plus the participant's fee, by providing his signature.
- the output of index 3 is only relevant for messages $\mathsf{a}(v \to \mathsf{B})$ where $v > 0$. Participant B can redeem v฿ from this output by providing his signature.

All transactions specify a lockTime $n + k$, where n is the current Bitcoin block number, and k is a positive constant. This ensures that a transaction can be mined only after k blocks. In this way, even if a transaction is signed by the arbiter and

sent to miners before the others, it has the same probability as the others of being appended to the blockchain.

To initialise the subchain, the arbiter puts the Genesis transaction on the Bitcoin blockchain. This transaction secures a small fraction of bitcoin, which can be redeemed by UR_1 through the arbiter signature. This value is then transferred to each subsequent update of the subchain (see Figure 4.4b). At each protocol stage, participants send incomplete UR transactions to the network. These transactions contain only in[0] and out[0], specifying the fee and the message for the subchain (including the value to be transferred).

Finally, to vote, meta-nodes add in[1], in[2] and out[2] to these transactions, to, respectively, put the required κ (from some transaction Stake_i), declare they want extend the last published update $\mathsf{Confirm}_{i-1}$, and specify the previous update to be rewarded. All the in[1] fields in a stage of the protocol must be different, to prevent attackers to vote more URs with the same funds.

4.5.1 Protocol performance

As seen in Section 4.2, the protocol runs in periods of duration Δ. Due to the mechanism for choosing the message to append to the subchain from the request pool, the protocol can publish at most one transaction per Bitcoin block. This means that a lower bound for Δ is the Bitcoin block interval ($\sim$10mins). To monitor the arbiter behaviour throughout protocol stages, it is expected that all meta-nodes share a coherent view of the request pool[9]. Then, Δ needs to be large enough to let each node synchronize the request pool with the rest of the network. A possible approach to cope with this issue is to make meta-nodes broadcast their voted updates, and to keep a list of other ones (considering only those which satisfy the format of transactions, as in Section 4.5). More efficient approaches could exploit distributed shared memories [44, 62].

[9] A naïve solution to achieve this goal, similar to that used by Bitcoin, requires meta-nodes to broadcast their request pool lists at each stage. A clear drawback of this solution is the significant overhead it adds to the network. Finding a more efficient alternative is still an open issue.

4.5.2 Overcoming the metadata size limit.

As noted in Section 4.5, we use `OP_RETURN` unspendable scripts to embed metadata in Bitcoin transactions. Since Bitcoin limits the size of such metadata to 80 bytes, this might not be enough to store the data needed by platforms. To overcome this issue, one can use distributed hash tables (like, for intance, *Kademlia* [70]) maintained by meta-nodes. In this way, instead of storing full message data in the blockchain, `OP_RETURN` scripts would contain only the corresponding message digests. The unique identifier of the Bitcoin transaction can be used as the key to retrieve the full message data from the hash table.

Chapter 5

Related work

In this chapter, we briefly expose some related work, presenting the state-of-the-art in the research field of behavioural contracts. In detail, we first present the research direction on formal models and architectures for contract-oriented computing, and then we examine some studies and implementations concerning the execution of contracts in decentralized environments, in particular referring to cryptocurrencies like Bitcoin and Ethereum.

Contract models and architectures. Our middleware builds upon CO_2 [29, 27], a core calculus for contract-oriented computing; in particular, the middleware implements all the main primitives of CO_2 (`tell`, `send`, `receive`), and it introduces new concepts, like e.g. the `accept` primitive, time constraints, and reputation.

From the theoretical viewpoint, the idea of constraint-based interactions has been investigated in other process calculi, e.g. Concurrent Constraint Programming (CCP [81]), and cc-pi [42], albeit the kind of interactions they induce is quite different from ours. In CCP, there is a global constraint store through which processes can interact by telling/asking constraints.

In cc-pi, interaction is a mix of name communication *à la* π-calculus [71] and `tell` *à la* CCP (which is used to put constraints on names). E.g., $\bar{x}\langle z\rangle$ and $y\langle w\rangle$ can synchronise iff the constraint store entails $x = y$; when this happens, the equality $z = w$ is added to the store, unless making it inconsistent. In cc-pi consistency plays a crucial role: `tell`s *restricts* the future interactions with other

processes, since adding constraints can lead to more inconsistencies; by contrast, in our middleware telling a contract *enables* interaction with other services, so consistency is immaterial.

The notion of time in behavioural contracts has been studied in [39], which addresses a timed extension of multi-party asynchronous session types [61]; however, the goals of [39] are quite different from ours. The approach pursued in [39] is top-down: a *global type* (specifying the overall communication protocol of a set of services, and satisfying some safety properties, e.g. deadlock-freedom) is projected into a set of *local types*; then, a composition of services preserves the properties of the global type if each service type-checks against the associated local type. Our middleware fosters a different approach to service composition: a distributed application is built bottom-up, by advertising contracts to delegate work to external (unknown and untrusted) services. Both our approach and [39, 75] use runtime monitoring to detect contract violations and assign the blame; additionally, in our middleware these data are exploited as an automatic source of information for the reputation system. Another formalism for communication protocols with time constraints is proposed in [56], where live sequence charts are extended with a global clock. The approaches in [39, 56] cannot be directly used in our middleware, because they do not provide algorithms to decide compliance, or to construct a contract compliant with a given one.

From the application viewpoint, several works have investigated the problem of *service selection* in open dynamic environments [8, 72, 90, 91]. This problem consists in matching client requests with service offers, in a way that, among the services respecting the given functional constraints, the one which maximises some *non-functional* constraints is selected. These non-functional constraints are often based on quality of service (QoS) metrics, e.g. cost, reputation, guaranteed throughput or availability, etc. The selection mechanism featured by our middleware does not search for the "best" contract compliant with a given one (actually, typical compliance relations in behavioural contracts are qualitative, rather than quantitative); the only QoS parameter we take into account is the reputation of services (see Section 3.2.2). In [91, 8] clients can require a sequence of tasks together with a set of non-functional constraints, and the goal is to find an assignment of tasks to services which optimises all the given constraints. There are two main differences between these approaches and ours. First, unlike behavioural

contracts, tasks are considered as atomic activities, not requiring any interaction between clients and services. Second, unlike ours, these approaches do not consider the possibility that a service may not fulfil the required task.

In the work [72], a service selection mechanism is implemented where functional constraints can be required in addition to QoS constraints: the first are described in a web service ontology, while the others are defined as requested and offered ranges of basic QoS attributes. A runtime monitor and a reputation system are also implemented, which, similarly to ours, help to marginalise those services which do not respect the advertised QoS constraints. Some kinds of QoS constraints cannot be verified by the service broker, so their verification is delegated to clients. This can be easily exploited by malicious participants to carry on *slandering attacks* to the reputation system [59]: an attacker could destroy another participant's reputation by involving it in many sessions, and each time declare that the required QoS constraints have been violated. In our middleware there is no need to assume participants trusted, as the verification of contracts is delegated to the middleware itself and to trusted third parties.

Smart contracts and blockchains. The idea of using Bitcoin and its blockchain as the basis for decentralized contracts has been explored by several recent works (see Section VIII in [40] for a brief survey). For instance, [7, 34] design protocols for secure multiparty computations and fair lotteries, and [53] proposes a protocol for Byzantine agreement which is secure when the hashing power of the adversary is strictly less than that of the honest participants.

On a more practical side, *Blockstore* [36] is a key-value database with `get/set` operations; *Namecoin* [74] is a censorship-resistant domain registration mechanism; *CounterParty* [50] extends Bitcoin with advanced financial operations (like e.g., creation of virtual assets, payment of dividends, *etc.*), by embedding its own messages in Bitcoin transactions. *Typecoin* [49] implements a peer-to-peer affine commitment on Bitcoin, by combining properties of the affine logic and the proof-carrying authorization, while *Catena* [87] proposes a protocol to efficiently agree on a log of application-specific statements managed by an adversarial server; in particular, Catena was developed concurrently and indipendently of our work and shares the same underlying insight as ours, although intended for a specific

domain of use.

Contracts on Ethereum. *Ethereum* [43] is a new cryptocurrency similar to Bitcoin, that allows for developing *general-purpose* contracts, interpreted by a decentralized virtual machine which runs over Ethereum nodes. In what follows we briefly describe the core feature of this platform.

Ethereum contracts are scripts, as in Bitcoin, but differently from the latter, the scripting languages of Ethereum is Turing-complete. Ethereum scripts are run by Ethereum nodes upon payment of a reward from the client. When a node completes the execution of a contract, it can claim the reward by broadcasting a transaction with the computed result. Before adding the claimant transaction to the blockchain, and consequently assigning him the reward, the other miners execute a special part of the contract script to verify the correctness of the provided result. Similarly to Bitcoin, invalid transactions (i.e., where the result of the computation does not pass the verification script) can be ignored. In this way, miners are incentivized to verify transactions because, in case one of their transactions in a mined block is invalidated, they would lose the associated fee. Therefore, the correct result of a computation is the one agreed upon by the majority of miners.

However, implementing contracts over Ethereum has several issues. First, programmers must write the whole program in one of the Ethereum languages (Serpent or Solidity), without exploiting existing legacy software or external non-Ethereum services (unless using a trusted oracle, or a compiler from other general purpose languages / DSLs, which at the time of this writing is far to be released). This constraint is required because, while computations carried on by a proper Ethereum virtual machine guarantee to produce correct results, this cannot be ensured for other kinds of computation. A further practical limitation is that, after a contract is deployed on the Ethereum network, it cannot be modified anymore; the only way to update it is to broadcast a new contract with the modified script, but the old version can still be used[1].

Besides these practical issues, contracts running over Ethereum are subject to the attacks described in [69]. These attacks exploit the fact that Ethereum miners suffer from the so-called *verifier's dilemma*, according to which they cannot rationally

[1] However, there exist some techniques to mark a contract as no longer callable.

decide whether to verify transactions or not. Whatever choice they make, honest miners are vulnerable of an attack. If a miner honestly follows the protocol by validating all transactions, then an adversary can impersonate a claimant and spam nodes with resource-intensive transactions. Since honest nodes will spend a significant amount of time to verify them, the adversary gains an advantage in the race for mining the next block and obtaining the associated fee. Otherwise, if miners choose to disobey the protocol and skip verification of resource-intensive transactions, then an adversary can claim the reward of a contract by broadcasting a transaction with a meaningless result. Since this transaction will not be verified, the adversary obtains the reward, and in conclusion the client has wasted his money for an incorrect answer.

Chapter 6

Conclusions

In Chapter 3, we have explored a new application domain for behavioural contracts, i.e. their use as interaction protocols in MOMs. In particular, we have developed a middleware where services can advertise contracts (in the form of timed session types, TSTs), and interact through sessions, which are created only between services with compliant contracts. To implement the middleware primitives, we have exploited the theory of TSTs in Section 2.1: in particular, a decidable notion of compliance between TSTs, a decidable procedure to detect when a TST admits a compliant one (and, if so, to construct it), and a decidable runtime monitoring.

We have validated our middleware through a series of experiments. The scalability tests (Section 3.4.1) seem to suggest that the performance of middleware is acceptable for up to $100K$ latent contracts. However, we feel that good performance can be obtained also for larger contract stores, for two reasons. First, in our experiments we have considered the pessimistic scenario where *all* latent contracts in the store are potentially compliant with a newly advertised one. Second, the current prototype of the middleware is sequential and centralised: parallelising the instances of the compliance checker, or distributing those of the middleware, would result in a performance boost. The experiments about the reputation system (Section 3.4.2) show that the middleware can relieve developers from dealing with misbehaviour of external services, and still obtain efficient distributed applications, which dynamically reconfigure themselves to foster the interaction among trustworthy services.

Although in this thesis we have focused on TSTs, the middleware only makes mild assumptions about the nature of contracts, e.g., that their observable actions are `send` and `receive`, and that they feature some notion of compliance with a sound (but not necessarily complete) verification algorithm. Hence, with minor efforts it would be possible to extend the middleware to support other contract models. For instance, communicating timed automata [38] (which are timed automata with unbounded communication channels) would allow for multi-party sessions, while session types with assertions [37], would allow for an explicit specification of the constraints among the values exchanged in sessions.

Besides the issues related to the expressiveness of contracts and to the scalability of their primitives (e.g., service binding and composition, runtime monitoring, *etc.*), we believe that also security issues should be taken into account: indeed, attackers could make a service sanctioned by exploiting discrepancies between its contracts and its actual behaviour. These mismatches are not always easy to spot; analysis techniques are therefore needed to ensure that a service will not be susceptible to this kind of attacks.

To cope with the limits of a centralized middleware, in Chapter 4 we have presented a protocol to reach consensus on subchains, i.e. chains of platform- dependent messages embedded in the Bitcoin blockchain. Our protocol incentivizes platform nodes to validate messages before appending them to the subchain, making economically disadvantageous for an adversary to append inconsistent messages. To do so, the protocol implements a Proof-of-stake, a mechanism similar to the Proof-of-Work of Bitcoin, but where the chances to append a message to the subchain depend on the money stake of nodes, rather than their computational power. Summarizing, platform nodes vote the client requests they consider consistent, by putting a bitcoin deposit on them, and — whenever a message is published on the Bitcoin blockchain and confirmed by subsequent updates — the deposit is paid back to the voters along with the fee generated by the client. The results of the security analysis of our protocol are deepened in Section 4.4.2. In particular, Theorem 4.4.1 essentially fixes a lower bound — which depends on protocol parameters — to the stake ratio (i.e., the percentage of stake over the total of the network) that an adversary should own to get an economical advantage for voting inconsistent updates.

Ensuring the reliability and consistency of subchain embedded in the Bitcoin blockchain makes it possible to extend the model of contracts supported by Bitcoin itself. In particular, contract-oriented applications and services can exploit subchains to store tamper-proof records of their contract execution, thus avoiding the need to trust a single entity (e.g., a private server) that assumes the role of the middleware. Although in Chapter 4 we only show how to encode LTSs in the blockchain, the idea can be easily extended to more general models of behavioural contracts, as briefly discussed in the following section.

6.1 Future work

The work presented in this dissertation paves the way for many possible improvements and new research directions. A first goal would be that of implementing a decentralized version of the middleware in Bitcoin, exploiting our protocol for consensus on subchains. We already developed a preliminary prototype of it[1] (with a small set of features), but it needs to be extended and tested in a real-world scenario. Observe that, in the decentralized settings, some middleware primitives can be directly specified in the contracts (e.g., the `accept`, the `send`, or the `receive`), since contract violations are monitored by the whole network of meta-nodes (*strong* decentralization). However, the middleware can anyway exist as a framework to facilitate this process. For example, it can use the blockchain only as a (public and immutable) storage of the traces generated by the contracts execution (*weak* decentralization).

These implementations both require an important re-organization of the architecture of the middleware, and an in-depth analysis of the new possible critical issues. In particular, the strong approach might have a heavy disadvantage in terms of performance, since a large amount of nodes have to execute the same computations to validate the new contract state (this is, essentially, the behaviour of Ethereum and of the native limited contract system of Bitcoin). Therefore, a challenging task could be that of finding suitable technical solutions to overcome these scalability limits.

A second interesting open problem concerns the role of the arbiter in our Proof-of-

[1] http://contractvm.github.io/

Stake protocol. We said in previous Section 4.3.3 that, in the protocol, the arbiter does not play the role of a trusted authority. This is a crucial fineness: if the arbiter was intended to be a trusted authority, it would partecipate in the consensus mechanism (i.e., voting and validating update requests). Conversely, the arbiter is only required to sign voted updates, but does not participate in the protocol. Since the request pool is public, nodes can verify that the arbiter actually signs all and only the voted updates, thus detecting any misbehaviour.

We have assumed the arbiter to be honest, to simplify our presentation. However, if the arbiter stops behaving honestly, nodes can easily fork the subchain and elect a new arbiter, but this could cause a significant overhead in the execution of the protocol, opening the way to potential Denial-of-Service attacks. For this reasons, we are currently working on an extension of the protocol in which the arbiter can be safely and quickly replaced in case of misbehaviours. The extension relies on some properties of the *Elliptic-curve Cryptography* (ECC), and its intuition consists essentially of rotating the key used by the arbiter to sign transactions, through a *secret sharing* mechanism that involves the nodes of the network. Of course, since this extension introduces additional points of vulnerability in the protocol, a wider security analysis needs to be done.

6.1.1 Contracts over subchains

The model of subchains defined in Section 4.1, based on LTSs, can be easily extended to model the computations of advanced contracts over the Bitcoin blockchain. A platform for contracts could exploit our model to represent the state of a contract as the state of the subchain, and model its possible state updates through the transition relation.

Implementing a platform for behavioural contracts would require a language for expressing them. To bridge this language with our abstract model, one can provide the language with an operational semantics, giving rise to an LTS describing the computations. Note that our assumption to model computations as a single LTS does not reduce the generality of the system, since a set of LTSs, each one modelling a contract, can be encoded in one LTS as their parallel composition. If the language is Turing-complete, an additional problem we would have to face is

the potential non-termination. This issue has been dealt with in different ways by different platforms. E.g., the approach followed by Ethereum [43] is to impose a fee for each instruction executed by its virtual machine. If the fee does not cover the cost of the whole computation, the execution terminates.

A usable platform must also allow to create new contracts at run-time. Since in our model the LTS representing possible computations is fixed, we would need a mechanism to "extend" it. To handle the publication of new contracts, we could modify the protocol so that UR may contain its code, and the unique identifier of the transaction also identifies the contract. In this extended model, update requests would also contain the identifier of the contract to be updated, so that meta-nodes can execute the corresponding code.

Bibliography

[1] Making sense of blockchain smart contracts. http://www.coindesk.com/making-sense-smart-contracts/. Last accessed 2017/10/22.

[2] opreturn.org. http://opreturn.org/. Last accessed 2017/11/10.

[3] PayPal buyer protection. https://www.paypal.com/us/webapps/mpp/ua/useragreement-full#13. Accessed: July 8, 2015.

[4] L. Aceto, A. Ingólfsdóttir, K. G. Larsen, and J. Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, New York, NY, USA, 2007.

[5] R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[6] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Fair two-party computations via Bitcoin deposits. In *Financial Cryptography Workshops*, pages 105–121, 2014.

[7] M. Andrychowicz, S. Dziembowski, D. Malinowski, and L. Mazurek. Secure multiparty computations on Bitcoin. In *IEEE S & P*, pages 443–458, 2014.

[8] D. Ardagna and B. Pernici. Adaptive service composition in flexible processes. *IEEE Trans. Software Eng.*, 33(6):369–384, 2007.

[9] N. Atzei and M. Bartoletti. Developing honest Java programs with Diogenes. In *Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*, pages 52–61, 2016.

[10] N. Atzei, M. Bartoletti, T. Cimoli, S. Lande, M. Murgia, A. S. Podda, and L. Pompianu. Contract-oriented programming with timed session types, Betty Book, 2017. http://tcs.unica.it/papers/

co2-middleware-tutorial.pdf.

[11] N. Atzei, M. Bartoletti, M. Murgia, E. Tuosto, and R. Zunino. Contract-oriented design of distributed applications: a tutorial. https://tcs.unica.it/papers/diogenes-tutorial.pdf, 2016.

[12] M. Babaioff, S. Dobzinski, S. Oren, and A. Zohar. On Bitcoin and red balloons. In *ACM Conference on Electronic Commerce (EC)*, pages 56–73, 2012.

[13] C. Baier and J. Katoen. *Principles of model checking*. MIT Press, 2008.

[14] W. Banasik, S. Dziembowski, and D. Malinowski. Efficient zero-knowledge contingent payments in cryptocurrencies without scripts. In *ESORICS*, volume 9879 of *LNCS*, pages 261–280. Springer, 2016.

[15] G. Banavar, T. D. Chandra, R. E. Strom, and D. C. Sturman. A case for message oriented middleware. In *Proc. Distributed Computing*, pages 1–18, 1999.

[16] F. Barbanera and U. de'Liguoro. Two notions of sub-behaviour for session-based client/server systems. In *PPDP*, pages 155–164, 2010.

[17] F. Barbanera and U. de'Liguoro. Sub-behaviour relations for session-based client/server systems. *Math. Struct. in Comp. Science*, pages 1–43, 1 2015.

[18] M. Bartoletti, T. Cimoli, M. Murgia, A. S. Podda, and L. Pompianu. Compliance and subtyping in timed session types. In *FORTE 2015*, pages 161–177, 2015.

[19] M. Bartoletti, T. Cimoli, M. Murgia, A. S. Podda, and L. Pompianu. A contract-oriented middleware. In *Proc. FACS*, 2015.

[20] M. Bartoletti, T. Cimoli, G. M. Pinna, and R. Zunino. Contracts as games on event structures. *J. Log. Algebr. Meth. Program.*, 85(3):399–424, 2016.

[21] M. Bartoletti, T. Cimoli, and R. Zunino. Compliance in behavioural contracts: a brief survey. In *PLABS*, volume 9465 of *LNCS*. Springer, 2015.

[22] M. Bartoletti, S. Lande, and A. S. Podda. A proof-of-stake protocol for consensus on bitcoin subchains. In *Workshop on Trusted Smart Contracts*, 2017.

[23] M. Bartoletti, M. Murgia, A. Scalas, and R. Zunino. Verifiable abstractions for contract-oriented systems. *JLAMP*, 86:159–207, 2017.

[24] M. Bartoletti and L. Pompianu. An analysis of Bitcoin OP_RETURN meta-

data. In *Financial Cryptography Workshops*, 2017. Also available as CoRR abs/1702.01024.

[25] M. Bartoletti and L. Pompianu. An analysis of Bitcoin OP_RETURN metadata. In *Financial Cryptography Workshops*, volume 10323 of *LNCS*. Springer, 2017.

[26] M. Bartoletti, A. Scalas, E. Tuosto, and R. Zunino. Honesty by typing. *Logical Methods in Computer Science*, 2017. Pre-print available as: https://arxiv.org/abs/1211.2609.

[27] M. Bartoletti, E. Tuosto, and R. Zunino. Contract-oriented computing in CO_2. *Sci. Ann. Comp. Sci.*, 22(1), 2012.

[28] M. Bartoletti, E. Tuosto, and R. Zunino. On the realizability of contracts in dishonest systems. In *COORDINATION*, 2012.

[29] M. Bartoletti and R. Zunino. A calculus of contracting processes. In *LICS*, 2010.

[30] M. Bartoletti and R. Zunino. Constant-deposit multiparty lotteries on Bitcoin. In *Financial Cryptography Workshops*, 2017. Also available as IACR Cryptology ePrint Archive 955/2016.

[31] G. Behrmann, A. David, and K. G. Larsen. A tutorial on Uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer, 2004.

[32] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *ACPN*, pages 87–124, 2003.

[33] I. Bentov, A. Gabizon, and A. Mizrahi. Cryptocurrencies without proof of work. In *Financial Cryptography Workshops*, volume 9604 of *LNCS*, pages 142–157. Springer, 2016.

[34] I. Bentov and R. Kumaresan. How to use Bitcoin to design fair protocols. In *CRYPTO*, volume 8617 of *LNCS*, pages 421–439. Springer, 2014.

[35] J. O. Blech, Y. Falcone, H. Rueß, and B. Schätz. Behavioral specification based runtime monitors for OSGi services. In *ISoLA*, pages 405–419, 2012.

[36] Blockstore: Key-value store for name registration and data storage on the Bitcoin blockchain. https://github.com/blockstack/blockstore, 2014.

[37] L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *CONCUR*, 2010.

[38] L. Bocchi, J. Lange, and N. Yoshida. Meeting deadlines together. In *CONCUR*, 2015. To appear.

[39] L. Bocchi, W. Yang, and N. Yoshida. Timed multiparty session types. In *CONCUR*, pages 419–434, 2014.

[40] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten. SoK: Research perspectives and challenges for Bitcoin and cryptocurrencies. In *IEEE S & P*, pages 104–121, 2015.

[41] A. Brogi, C. Canal, and E. Pimentel. Behavioural types for service integration: Achievements and challenges. *ENTCS*, 180(2):41–54, 2007.

[42] M. G. Buscemi and U. Montanari. CC-Pi: A constraint-based language for specifying service level agreements. In *ESOP*, pages 18–32, 2007.

[43] V. Buterin. Ethereum: a next generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper, 2013.

[44] M. Cai, A. Chervenak, and M. Frank. A peer-to-peer replica location service based on a distributed hash table. In *ACM/IEEE Conference on High Performance Networking and Computing*, page 56. IEEE Computer Society, 2004.

[45] G. Castagna, M. Dezani-Ciancaglini, E. Giachino, and L. Padovani. Foundations of session types. In *PPDP proceedings*, pages 219–230, 2009.

[46] G. Castagna, N. Gesbert, and L. Padovani. A theory of contracts for Web services. *ACM Transactions on Programming Languages and Systems*, 31(5), 2009.

[47] L. Y. Chen and H. P. Reiser, editors. *Distributed Applications and Interoperable Systems - 17th IFIP WG 6.1 International Conference, DAIS 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings*, volume 10320 of *Lecture Notes in Computer Science*. Springer, 2017.

[48] K. Crary and M. J. Sullivan. Peer-to-peer affine commitment using Bitcoin. In *ACM PLDI*, pages 479–488, 2015.

[49] K. Crary and M. J. Sullivan. Peer-to-peer affine commitment using bitcoin. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 479–488, New York, NY, USA, 2015. ACM.

[50] R. Dermody, A. Krellenstein, O. Slama, and E. Wagner. Counter-Party: Protocol specification. http://counterparty.io/docs/protocol_specification/, 2014.

[51] C. Dwork and M. Naor. Pricing via processing or combatting junk mail. In *CRYPTO*, volume 740 of *LNCS*, pages 139–147. Springer, 1993.

[52] I. Eyal and E. G. Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography and Data Security*, volume 8437 of *LNCS*, pages 436–454. Springer, 2014.

[53] J. A. Garay, A. Kiayias, and N. Leonardos. The Bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT*, volume 9057 of *LNCS*, pages 281–310. Springer, 2015.

[54] S. J. Gay and V. T. Vasconcelos. Linear type theory for asynchronous session types. *J. Funct. Program.*, 20(1):19–50, 2010.

[55] J. Göbel, P. Joschko, A. Koors, and B. Page. The discrete event simulation framework DESMO-J: review, comparison to other frameworks and latest development. In *European Conference on Modelling and Simulation (ECMS)*, pages 100–109. European Council for Modeling and Simulation, 2013.

[56] D. Harel and R. Marelly. Playing with time: On the specification and execution of time-enriched LSCs. In *MASCOTS*, pages 193–202, 2002.

[57] R. Heckel and M. Lohmann. Towards contract-based testing of Web services. *Electr. Notes Theor. Comput. Sci.*, 116:145–156, 2005.

[58] A. Hern. A history of Bitcoin hacks. http://www.theguardian.com/technology/2014/mar/18/history-of-bitcoin-hacks-alternative-currency, march 2014.

[59] K. J. Hoffman, D. Zage, and C. Nita-Rotaru. A survey of attack and defense techniques for reputation systems. *ACM Comput. Surv.*, 42(1), 2009.

[60] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type

disciplines for structured communication-based programming. In *ESOP*, 1998.

[61] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL*, 2008.

[62] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. In *PODC*, pages 213–222. ACM, 2002.

[63] A. Kiayias, I. Konstantinou, A. Russell, B. David, and R. Oliynykov. Ouroboros: A provably secure Proof-of-Stake blockchain protocol. *IACR Cryptology ePrint Archive*, 2016:889, 2016.

[64] A. Kiayias, H. Zhou, and V. Zikas. Fair and robust multi-party computation using a global transaction ledger. In *EUROCRYPT*, pages 705–734, 2016.

[65] R. Kumaresan and I. Bentov. How to use Bitcoin to incentivize correct computations. In *ACM CCS*, pages 30–41, 2014.

[66] R. Kumaresan, T. Moran, and I. Bentov. How to use Bitcoin to play decentralized poker. In *ACM CCS*, pages 195–206, 2015.

[67] C. Laneve and L. Padovani. The *must* preorder revisited. In *CONCUR*, LNCS 4703, pages 212–225. Springer, 2007.

[68] J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Trans. Inf. Syst. Secur.*, 12(3), 2009.

[69] L. Luu, J. Teutsch, R. Kulkarni, and P. Saxena. Demystifying incentives in the consensus computer. In *ACM CCS*, pages 706–719. ACM, 2015.

[70] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Workshop on Peer-to-Peer Systems (IPTPS)*, volume 2429 of *LNCS*, pages 53–65. Springer, 2002.

[71] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1), 1992.

[72] A. Mukhija, A. Dingwall-Smith, and D. Rosenblum. QoS-aware service composition in Dino. In *ECOWS*, pages 3–12, 2007.

[73] S. Nakamoto. Bitcoin: a peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.

[74] Namecoin: a decentralized DNS service. https://wiki.namecoin.org, 2011.

[75] R. Neykova, L. Bocchi, and N. Yoshida. Timed runtime monitoring for multiparty conversations. In *BEAT*, pages 19–26, 2014.

[76] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *CAV*, pages 376–398, 1991.

[77] 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.

[78] M. Rosenfeld. Analysis of hashrate-based double spending. *CoRR*, abs/1402.2009, 2014.

[79] T. Ruffing, A. Kate, and D. Schröder. Liar, liar, coins on fire!: Penalizing equivocation by loss of Bitcoins. In *ACM CCS*, pages 219–230, 2015.

[80] A. Sahai, V. Machiraju, M. Sayal, A. P. A. van Moorsel, and F. Casati. Automated SLA monitoring for Web services. In *DSOM*, pages 28–41, 2002.

[81] V. A. Saraswat and M. C. Rinard. Concurrent constraint programming. In *POPL*, pages 232–245, 1990.

[82] S. Sebastio and A. Vandin. MultiVeStA: statistical model checking for discrete event simulators. In *Proc. ValueTools*, pages 310–315, 2013.

[83] M. Srivatsa, L. Xiong, and L. Liu. TrustGuard: countering vulnerabilities in reputation management for decentralized overlay networks. In *WWW*, pages 422–431, 2005.

[84] A. Strunk. QoS-aware service composition: A survey. In *ECOWS*, pages 67–74. IEEE, 2010.

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

[86] K. Takeuchi, K. Honda, and M. Kubo. An interaction-based language and its typing system. In *PARLE*, pages 398–413, 1994.

[87] A. Tomescu and S. Devadas. Catena: Efficient non-equivocation via Bitcoin. In *IEEE Symp. on Security and Privacy*, 2017.

[88] E. Tuosto. Contract-oriented services. In *WS-FM*, pages 16–29, 2012.

[89] M. T. B. Waez, J. Dingel, and K. Rudie. A survey of timed automata for the development of real-time systems. *Computer Science Review*, 9:1 – 26, 2013.

[90] T. Yu, Y. Zhang, and K.-J. Lin. Efficient algorithms for Web services selection with end-to-end QoS constraints. *ACM Transactions on the Web*, 1(1):6, 2007.

[91] L. Zeng, B. Benatallah, A. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-aware middleware for Web services composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, 2004.