UNIVERSITÀ DEGLI STUDI DI CAGLIARI

Facoltà di Scienze

Corso di Laurea Magistrale in Informatica

A general-purpose decentralized system based on the Bitcoin block-chain

Supervisor
Prof. Massimo Bartoletti

M.Sc. Candidate
Davide Gessa
Matr. N. 49196

ACADEMIC YEAR 2014/2015

Abstract

Today, many Internet services are implemented as centralized software, where the service, usually owned by a single corporation, acts as a trusted third party that handles user interactions.
The centralized approach relies on the assumption that users completely trust the central authority; this could become a single point of failure for the entire system, and so the central authority must be carefully designed.
In the past few years the *decentralized model* has gained popularity thanks to projects like *Bitcoin* or *Ethereum*. In this model, a service is implemented as a peer-to-peer network, where each node holds a public immutable data structure that maintains the historical data of all client transactions. So, if a node of the decentralized system is not available anymore, any other running node can take its place without losing of data and without interrupting the service.
We design and develop a general-purpose decentralized system based on the Bitcoin blockchain; our system is extensible through a plugin infrastructure that allows to implement arbitrary decentralized applications. To validate the general applicability of our system, we develop four case studies, also including a decentralized implementation of an existing contract-oriented middleware based on timed session types.

Contents

Chapter 1

Introduction

Today, many internet services are often implemented as centralized software, where the service, usually owned by a single corporation, acts as a trusted third party that handles user interactions. Some examples of the centralized paradigm are a financial institution that processes user transactions, an auction marketplace or an email service.

The centralized approach relies on the assumption that users completely trust the central authority, which must provide correct information and must be always available. This could become a single point of failure for the entire system, and so the central authority must be carefully designed. Furthermore, a centralized system could act maliciously, or it could act illegally. Another problem is that a centralized service could be made inaccessible by a censorship action or by an attacker.

1.1 Decentralized Applications

In the past few years the *decentralized model* has gained popularity thanks to projects like *Bitcoin* [19] or *Ethereum* [12]. In this model, a service is implemented as a peer-to-peer network, where each node acts at the same time as a service provider and as a service client. Instead of delegating crucial operations to a trusted authority, a decentralized system is based on cryptographic proofs (like e.g. the *proof of work* system of Bitcoin [19]). Another feature of this model is that each node holds a public immutable data structure that maintains the historical data of all client transactions. So, if a node of the decentralized system is not available anymore, any other running node can take its place without losing data, and without interrupting the service.

A notable example of a decentralized application is *Bitcoin* [19], a peer-to-peer electronic cash system. Bitcoin proposes a financial system where all transactions are stored in a public ledger, called *block-chain*; the block-chain is updated with new blocks of transactions, using a distributed consensus process called *mining*, that enforces a chronological order in the block-chain, protects the neutrality of the network, and allows different computers to agree on the state of the system. To be confirmed, transactions must be packed in a block that fits given cryptographic rules, verified by the network. These rules prevent previous blocks from being modified, since a change would invalidate all the following blocks. Mining also creates the equivalent of a "competitive lottery", that prevents any

individual from easily adding new blocks consecutively in the block chain. This way, no one can control what is included in the block-chain, or replace parts of the block chain to roll back their own spends.

The idea of the Bitcoin block-chain has been used as the foundation of several decentralized applications [8, 20, 18, 9, 12]. For instance, *Namecoin* [8] uses transactions to create a registration mechanism for domains. Namecoin's flagship use case is the censorship-resistant top level domain .bit, which is functionally similar to .com or .net domains, but independent from *ICANN*, the main governing body for domain names. *CounterParty* [18] is a decentralized application for creating peer-to-peer financial products on the Bitcoin block-chain; its protocol is primarily designed to create virtual assets, and to issue dividends. The CounterParty protocol defines a set of rules for embedding messages inside block-chain transactions, forcing all Bitcoin nodes to download CounterParty messages even if they do not need these data. *Blockstore* [9] is a key-value data storage built on top of the Bitcoin block-chain; the block-chain is still used, but only for storing meta-data. Full data are stored in a distributed hash table (DHT); this implies that message size is unlimited, and only the interested nodes download the full data.

While the projects outlined before focus on specific application domains, *Ethereum* [12] allows for developing *general-purpose* decentralized applications (*dapp*) using a Turing-complete language. To enable the execution of these user-created applications, the Ethereum daemon includes a virtual machine. After a dapp is submitted the Ethereum block-chain, it cannot be stopped by anyone, it is not alterable by a third party, and its code is executed in Ethereum nodes. A dapp is activated and executed by the decentralized virtual machine, every time that a user sends a transaction that contains a call to the dapp. The transaction must include a certain amount of electronic cash, to pay all nodes for its execution.

Despite its generality, Ethereum has some drawbacks: a dapp must be written using one of the Ethereum languages, and it cannot integrate external software or API easily, because the code execution must provide the same result on every node. Another limitation is that a node cannot restrict the set of dapp it runs. After a dapp is deployed in the Ethereum network, it cannot be modified anymore, because each dapp is associated with its hash value in the block-chain: the only way to update it is to broadcast a new dapp with the modified code, but the old version remains usable.

Note that, since nodes are distributed over an open network, they could be run by attackers who try to subvert legitimate computations. In Ethereum, clients can only protect themselves against this kind of attacks either by locally running a node, or by connecting to a trusted node.

A further disadvantage of Ethereum is that clients need to run the block-chain daemon besides the application. This is quite impractical, especially when the computing device has limited resources (like e.g. power, bandwidth, disk space). Indeed, today the Ethereum block-chain uses already about 1 GB of disk space and needs about 2 hours for synching.

1.2 Contributions

In this thesis we design and develop a general-purpose decentralized system based on the Bitcoin block-chain. Users interact with the system by sending cryptographic signed messages to the block-chain. This guarantees that messages cannot be deleted or modified by malicious users; furthermore, users can obtain a proof of existence for the messages sent to the block-chain (so to prevent e.g. their repudiation). To overcome the limitation to 40 bytes per transaction imposed by the Bitcoin protocol, in our system we store messages in a data structure which combines the block-chain with a DHT.

The proposed system is extensible through a plugin infrastructure that allows to implement arbitrary decentralized applications. Our system saves in the block-chain only the messages exchanged by clients; so, we depart from Ethereum, which uses the block-chain also to save the state of computations. This design choice is crucial, since it allows us to overcome the drawbacks of Ethereum outlined before. In practice, each node in our system can choose which plugin to execute; further, we do not impose constraints on the programming language used to develop plugins, which can invoke legacy applications or external services in case they need to.

Our system exploits a consensus mechanism to protect clients in case some nodes are governed by attackers. Specifically, when a client queries a plugin, the query is replicated and sent to a set of distinct nodes; the actual result is obtained by taking the majority of the received answers. This mechanism is completely transparent to client programmers, which exploit the functionalities of plugins via a set of APIs. A client application does not need to run the block-chain daemon anymore, because the consensus mechanism is implemented directly in the client library.

To validate the general applicability of our system, we develop four case studies. Each of them includes a plugin, the corresponding client library, and some usage examples. The first case study is a simple *HelloWorld* plugin, where clients can save their names and get previously saved names (with the number of their occurrences). The main purpose of this toy example is to present a walk-through of how a plugin and its client library are written. The second case study implements a key-value decentralized database, similar to the one provided by Blockstore [9]. The third case study is a *FIFO* message oriented middleware which acts as a decentralized broker for messages. With this plugin clients can produce or consume messages from queues, in a way similar to the *RabbitMQ middleware* [6]. The last case study is far more complex, as it implements a decentralized contract-oriented middleware based on timed session types presented in [10]. The middleware uses contracts to allow interactions between distributed services; it handles the composition of services, monitoring their interactions to detect contract violations.

All produced software, including the node daemon with all case studies, the client library and the DHT implementation, are available as open-source projects on [13, 14, 15].

1.3 Structure of the thesis

The thesis is composed of 8 chapters, including this introduction; Part I examines the background behind this thesis, focusing on contracts and timed session types in Chapter 2. Part II explains how our decentralized system works; Chapter 3 shows an overview of how the Bitcoin block-chain is extended for data storage of messages. Then, Chapter 4 focuses on the system architecture, detailing the functions of each module. In chapter 5, the client library that allows to write client applications using this system is proposed. In Chapter 6 a list of attack sceneries are enumerated, describing how these situations are handled, and which are the effects both in a centralized and in our decentralized system. In Chapter 7 we validate the general applicability of our system, exploiting it to write the above-mentioned four case studies, including the contract-oriented middleware based on timed session types (Section 7.4). An overview of conclusions, related and future works are finally given in Chapter 8.

Part I

Background

Chapter 2

Contracts and timed session types

2.1 Overview on contracts

In computer science, contracts are interacting processes with an explicit notion of obligations and objectives, and are being developed to regulate the communication between distributed components, hereafter called participants (each one with possibly conflicting individual goals and not necessarily honest), belonging to potentially distinct systems and infrastructures, and under the control of different providers.

Over the last few years several formal representations for contracts have been described in literature, such as event structures and session types. In this chapter we provide a brief summary of a session types variant that introduces timing capabilities.

2.2 Timed Session Types

In this section we recall from [10] some useful notions about the theory of timed session types.

2.2.1 Semantics of Timed Session Types

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 2.2.4). This relation describes all and only the *correct* interactions: for instance, 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.

We denote with $\mathbb{V} = \mathbb{C} \to \mathbb{R}_{\geq 0}$ the set of clock valuations (ranged over by $\nu, \eta, \ldots$), and with ν_0 the valuation mapping each clock to zero. We write $\nu + \delta$ for the valuation which increases ν by δ, i.e., $(\nu + \delta)(t) = \nu(t) + \delta$ for all $t \in \mathbb{C}$. For a set $R \subseteq \mathbb{C}$, we write $\nu[R]$ for the *reset* of the clocks in R, i.e.,

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

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

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

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

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

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

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

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

Definition 2.2.1 (Semantics of guards) *For all guards g, we define the set of clock valuations $[\![g]\!]$ inductively as follows, where $\circ \in \{<, \leq, =, \geq, >\}$:*

$$[\![g]\!] = \begin{cases} \mathbb{V} & \text{if } g = true \\ \mathbb{V} \setminus [\![g']\!] & \text{if } g = \neg g' \\ [\![g_1]\!] \cap [\![g_2]\!] & \text{if } g = g_1 \wedge g_2 \\ \{\nu \mid \nu(t) \circ d\} & \text{if } g = t \circ d \\ \{\nu \mid \nu(t) - \nu(t') \circ d\} & \text{if } g = t - t' \circ d \end{cases}$$

Before defining the semantics of TSTs, we recall from [11] some basic operations on *sets* of clock valuations (ranged over by $\mathcal{K}, \mathcal{K}', \ldots \subseteq \mathbb{V}$).

Definition 2.2.2 (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 2.2.3 *For all TSTs p, we define the set of clock valuations $rdy(p)$ as:*

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

Definition 2.2.4 (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 *$[!a\{g,R\}]\,p$. The semantics of TSTs is defined as a labelled relation $\rightarrow$ over configurations, whose labels are either silent actions τ, delays δ, or branch labels. As usual, we denote with $\rightarrow^*$ the reflexive and transitive closure of the relation $\rightarrow$.*

We now comment the rules in Figure 2.1. The first four rules are auxiliary, as they describe the behaviour of a TST in isolation. Rule [+] allows a TST to commit to the branch $!a$ of her internal choice, provided that the corresponding guard is satisfied in the clock valuation ν. This results in the term $[!a\{g, R\}]\, p$, which represents the fact that the endpoint has committed to branch $!a$ in a specific time instant: actually, it can only fire $!a$ through rule [!] (which also resets the clocks in R), while time cannot pass. 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 *rdy* in Figure 2.1. The last three rules deal with configurations of two TSTs. Rule [S-+] allows a TSTs to commit in an internal choice. Rule [S-τ] is the standard synchronisation rule *à la* CCS; note that B is assumed to read a message as soon as it is sent, so A never blocks on internal choices. Rule [S-Del] allows time to pass, equally for both endpoints.

Example 2.2.5 *Let* $p = !a + !b\{t > 2\}$, *let* $q = ?b\{t > 5\}$, *and consider the following computations:*

$$\begin{aligned}(p, \nu_0) \mid (q, \eta_0) &\xrightarrow{7}\xrightarrow{\tau} ([!b\{t > 2\}], \nu_0 + 7) \mid (q, \eta_0 + 7) \\ &\xrightarrow{\tau} (\mathbf{1}, \nu_0 + 7) \mid (\mathbf{1}, \eta_0 + 7) \qquad (2.1)\end{aligned}$$

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

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

The computation in (2.1) *reaches success, while the other two computations reach the deadlock state. In* (2.2), p *commits to the choice* $!a$ *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* $?a$*). In* (2.3), p *commits to* $!b$ *after* 3 *time units; here, the rightmost endpoint would offer* $?b$*, — 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* $(!b\{t > 2\}, \nu_0) \mid (q, \eta_0)$ *never reaches deadlock — contradicting our intuition that these endpoints should not be considered compliant.*

Definition 2.2.6 (Compliance [10]) *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 the labels of* p *and* q *belong to the same context, and:*

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

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

2.2.2 Dual construction

The dual construction makes sense only for those TSTs for which a compliant exists. To this purpose, we define a procedure (more precisely, a kind system) which computes the set of clock valuations $\mathcal{K}$ (called *kinds*) such that p admits a compliant TST in all $\nu \in \mathcal{K}$.

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

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

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

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

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

Figure 2.2: Kind system for TSTs.

Definition 2.2.7 (Kind system) *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 compliant in every ν: indeed, $\mathbf{1}$ is compliant with itself. The kind of an exernal 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). In order to obtain completeness of the kind system we need the greatest fixed point.

The following theorem states that every TST is kindable. We stress the fact that being kindable does not imply admitting a compliant. This holds if and only if ν_0 belongs to the kind (see Theorems 2.2.11 and 2.2.12).

Theorem 2.2.8 *For all closed p, there exists some $\mathcal{K}$ such that $\vdash p : \mathcal{K}$.*

The following theorem states that the problem of determining the kind of a TST is decidable. This might seem surprising, as the cardinality of kinds is $2^{2^{\aleph_0}}$. However, the kinds constructed by our inference rules can always be represented syntactically by guards [16].

Theorem 2.2.9 *Kind inference is decidable.*

Definition 2.2.10 (Dual of a TST [10]) *For all kindable p kindable and kinding environments Γ, we define the TST $\mathrm{co}_\Gamma(p)$ (in short, $\mathrm{co}(p)$ when $\Gamma = \emptyset$)*

$$\begin{array}{rcll}
\mathrm{co}_\Gamma(\mathbf{1}) & = & \mathbf{1} & \\
\mathrm{co}_\Gamma(\&_{i\in I} ?a_i\{g_i, T_i\} \,.\, p_i) & = & \sum_{i\in I} !a_i\{g_i \wedge \mathcal{K}_i[T_i]^{-1}, T_i\} \,.\, \mathrm{co}_\Gamma(p_i) & \text{if } \Gamma \vdash p_i : \mathcal{K}_i \\
\mathrm{co}_\Gamma(\sum_{i\in I} !a_i\{g_i, T_i\} \,.\, p_i) & = & \&_{i\in I} ?a_i\{g_i, T_i\} \,.\, \mathrm{co}_\Gamma(p_i) & \\
\mathrm{co}_\Gamma(X) & = & X & \text{if } \Gamma(X) \text{ defined} \\
\mathrm{co}_\Gamma(\operatorname{rec} X \,.\, p) & = & \operatorname{rec} X \,.\, \mathrm{co}_{\Gamma\{^{\mathcal{K}}/_X\}}(p) & \text{if } \Gamma \vdash \operatorname{rec} X \,.\, p : \mathcal{K}
\end{array}$$

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

Theorem 2.2.11 (Soundness) *If* $\vdash p : \mathcal{K}$ *and* $\nu \in \mathcal{K}$, *then* $(p, \nu) \bowtie (\mathrm{co}(p), \nu)$.

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

2.2.3 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]. 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 2.2.1) deals with the case of honest participants; however, unlike the semantics in Section 2.2.1, 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 2.2.13 (Monitoring semantics of TSTs) Monitoring configurations $\gamma, \gamma', \ldots$ *are terms of the form* $P \,\|\, Q$, 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* $\lambda, \lambda', \ldots \in (\{\mathsf{A}, \mathsf{B}\} \times L) \cup \mathbb{R}_{\geq 0}$, *is defined in Figure 2.3.*

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

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

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

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

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

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

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

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

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

Figure 2.3: Monitoring semantics (symmetric rules omitted).

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 2.2.14 (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 [10].

Example 2.2.15 *Let $p = !a\{2<t<4\}$ be the TST of participant* A*, and let $q = ?a\{2<t<5\}+?b\{2<t<5\}$ be that of* B*. Participant* A *declares that she will send a between 2 and 4 time unit (abbr. t.u.), while* B *declares that he is willing to receive a or b if they are sent within 2 and 5 t.u. 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 $\eta=\langle 1.2,\ \mathsf{A}:!a,\ \mathsf{B}:?a\rangle$. 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]*: $\gamma_0 \xrightarrow{1}\!\!\!\!\rightarrow\xrightarrow{\mathsf{A}:!a}\!\!\!\!\rightarrow (\mathbf{0},[],\nu_0+1)\parallel(q,[],\eta_0+1)$, 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.*

Part II

General-purpose decentralized system

Chapter 3

Extending the bitcoin block-chain

One of the key features of this system is the storage of messages. In this Chapter we provide some reasonable motivations to avoid the usage of the block-chain for storing all the message data. Therefore, we compose the block-chain with a distributed hash table, exploiting the block-chain only for message meta-data.

This guarantees that messages cannot be deleted or modified by malicious users; furthermore, users can obtain a proof of existence for the messages sent to the block-chain (so to prevent e.g. their repudiation). To achieve these requirements, we exploited the block-chain technology introduced by Satoshi Nakamoto in *Bitcoin* [19] using it as an immutable data-structure.

Even if bitcoin-like block-chains are primarily intended to handle digital cash transactions, the protocol allows to include extra data in transactions using an `OPRETURN` script. The maximum amount of additional data in a transaction cannot exceed a limit of 40 bytes (or 80 bytes in newest protocol versions); this limit will avoid spam transactions in the block-chain, since transaction data are stored in all bitcoin nodes. To overcome this limit, we use the block-chain only to save message meta-data, while we use a distributed hash table to store the full data. The complete data model scheme with block-chain and DHT is illustrated in Figure 3.1.

3.1 Distributed Hash Table

A distributed hash table (DHT) is a class of a decentralized and distributed system that provides a service similar to an hash table: (key, value) pairs are stored in a DHT, and any participating node can efficiently retrieve the value associated with a given key. Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. This allows a DHT to scale to extremely large numbers of nodes and to handle continual node arrivals, departures, and failures.

In our prototype, we have developed a *Kademlia* distributed hash table, the one used by softwares like *eMule* [3] and *bittorrent* [2]. All stored messages are encoded in JSON format.

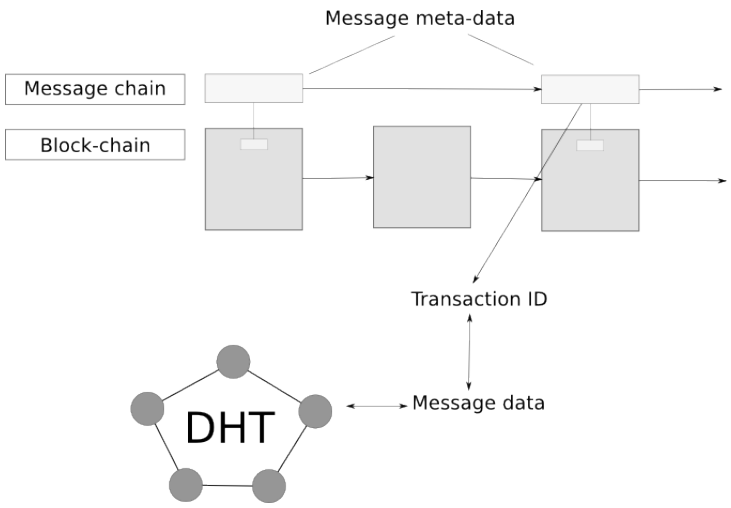

Figure 3.1: Data model with DHT and Block-chain

3.2 Message meta-data

As we said earlier, a transaction has only a small amount of bytes for storing other data, and we use these bytes for meta-data storing. To ensure that all messages meta-data will be stored in a transaction, we choose to use the oldest limit of 40 bytes for backward compatibility; the bytes allocation of the proposed protocol is detailed in Figure 3.2. A single message meta-data includes various fields, like a magic flag for message recognition, the hash of the message, plugin code and addressing information for retriving the correct message from the DHT.

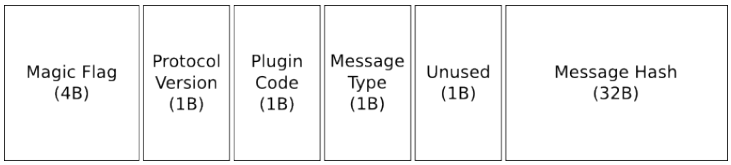

Figure 3.2: Format of OPRETURN message meta-data

The addressing information should be unique in the entire system; the ID of the block-chain transaction that contains meta-data is used as unique ID for the message. An hash of the DHT data is also included to avoid data modification performed by attackers. Another useful information is the sender of the message; this is achieved easily because the transaction is signed with the private key of the sender of the bitcoin transaction. The plugin informations are used by the plugin system to handle different types of messages.

Chapter 4

Decentralized system

In this Chapter we explain how the decentralized system works, focusing on the node daemon software and its modules.

The decentralized system is composed of a network of inter-connected nodes, where each node has the same capabilities of the others. A generic node is connected to the block-chain for storing meta-data and to the DHT for broadcasting and retrieving message data. The nodes offer services to client applications through an API interface; usually a client application is connected to more than one node, using data received from multiple nodes to establish which is the correct value through a majority mechanism.

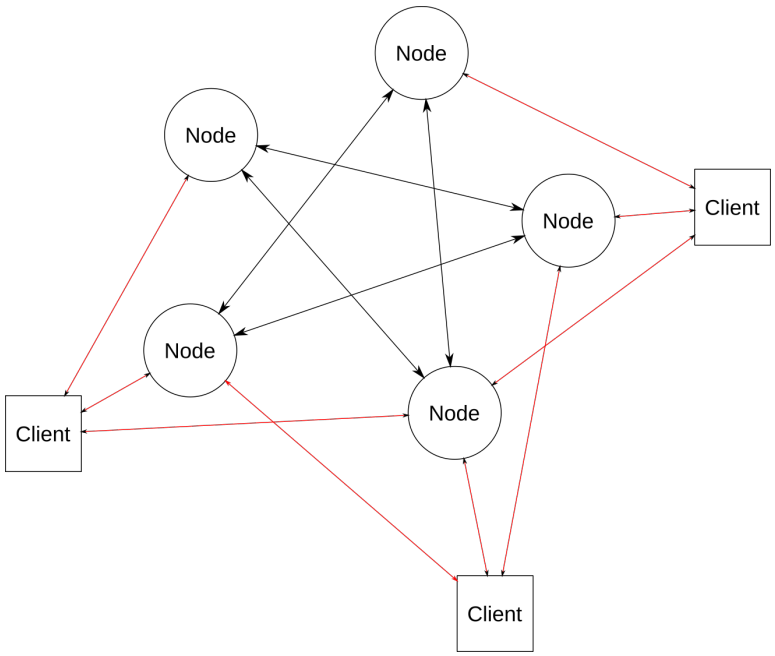

Figure 4.1: Example node network with clients connected

The node daemon is built on top of the *bitcoin-core* daemon (or similar); it communicates with the bitcoin-core through the standard json-rpc interface. The daemon polls the bitcoin-core for new blocks using the *Chain* module described in Section 4.1; if a new block is inserted in the block-chain, the Chain module retrieves all the transactions

included in the block and starts to scan for a new message. If a new message is found in a transaction identified by a transaction ID, the DHT is queried for the key correspondent of the transaction ID; after the message data has been received, it is checked for validity (hash, signer and size match). If the message is valid, it is passed to the *PluginManager* module described in Section 4.2, that finds a suitable plugin (four case study plugins are provided in Chapter 7).

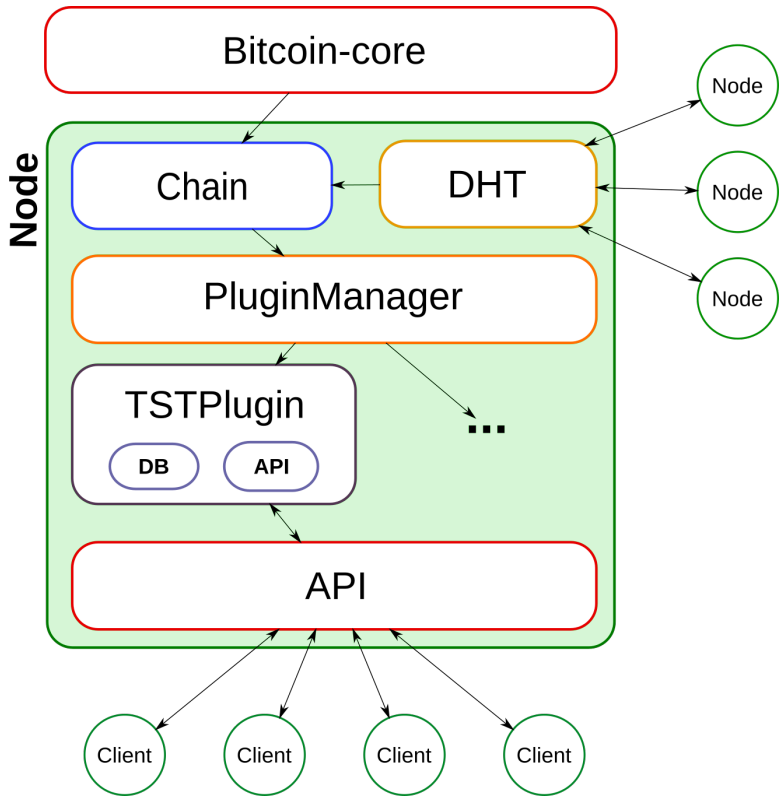

Figure 4.2: Node software modules

A client application that uses this system exploits the API described in Section 4.3 to perform queries and to prepare and broadcast messages. When a client application needs to broadcast a message, the client follows the flow illustrated in Figure 4.3: first, it sends the raw message to the node that will create an unsigned transaction for the given message; then the message is signed by the client. Finally, the node will broadcast the transaction to the block-chain, and the message data to the DHT.

4.1 Chain and DHT modules

The Chain module polls the bitcoin-core RPC to detect the presence of a new block; when a new block appears, the module retrieves the new block and scans it for transactions that contain messages. All found messages are checked for validity (protocol version and plugin availability); if all these information are OK, it calls the DHT module with a *findKey* query using the transaction ID as key. If the key is present in the DHT, the message data is downloaded and the message hash is compared with the meta-data hash; if they match, the message is marked as valid, and it is passed to the PluginManager module

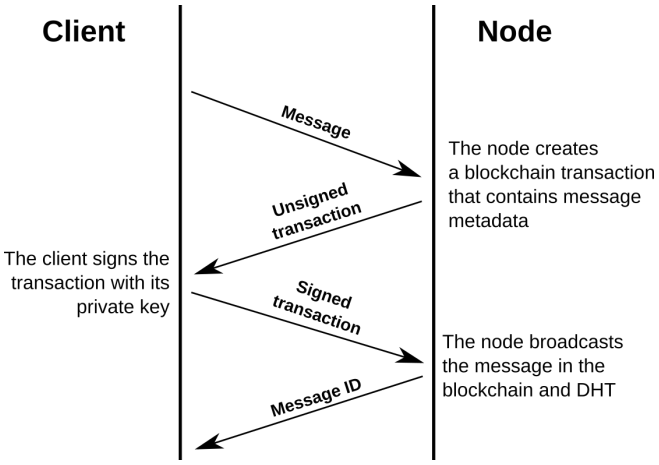

Figure 4.3: Client - Node message flow

for processing. If the key is not present in the DHT, the system tries again the *findKey* query until a timeout occurs.

4.2 PluginManager and Plugins modules

The node daemon also implements an abstraction layer, the PluginManager; through this module, it is possible to define arbitrary decentralized systems by defining a plugin script. When a new message arrives to the PluginManager, it selects the correct Plugin using the *plugin code* and the *type* fields in message meta-data. If a suitable plugin is present, the new message is passed to the corresponding *Plugin.handleMessage()* method.

In Chapter 7 four plugin are proposed, including a walk-through that explains how a plugin is written.

4.3 API module

To use the services offered, the system provides an API module that uses the JSON-RPC protocol; the API calls provide basic plugin-independent functions. Other API calls can be added using plugins, as explained in Section 4.2.

The plugin-independent API calls are:

- `info ()`: returns generic node information
- `broadcast (data)`: broadcasts a chain transaction
- `net.peers ()`: returns the list of connected peers
- `net.connections ()`: returns the number of connections

Chapter 5

Client library

In this Chapter we give details on how the client library works and how it connects to multiple node daemons with the purpose of enabling the consensus mechanism.

The decentralized client library is more complex than a centralized version. As we told before, each application using this system will connect to more than one node daemon in the aim to enable the majority mechanism for data validation. The client library is also integrated with a bitcoin wallet so that messages could be signed with a private key.

The plugin-agnostic library does not implement any Plugin; four examples of plugins with their client libraries are shown in Chapter 7.

The client library is divided in 2 core modules, the *ConsensusManager* and the *Wallet*. The library itself is useless without a plugin module; each plugin implements a custom module that handles plugin related API calls. For example, as illustrated in Figure 5.1, the module *ContractManager* acts as client interface for the TST plugin (Section 7.4).

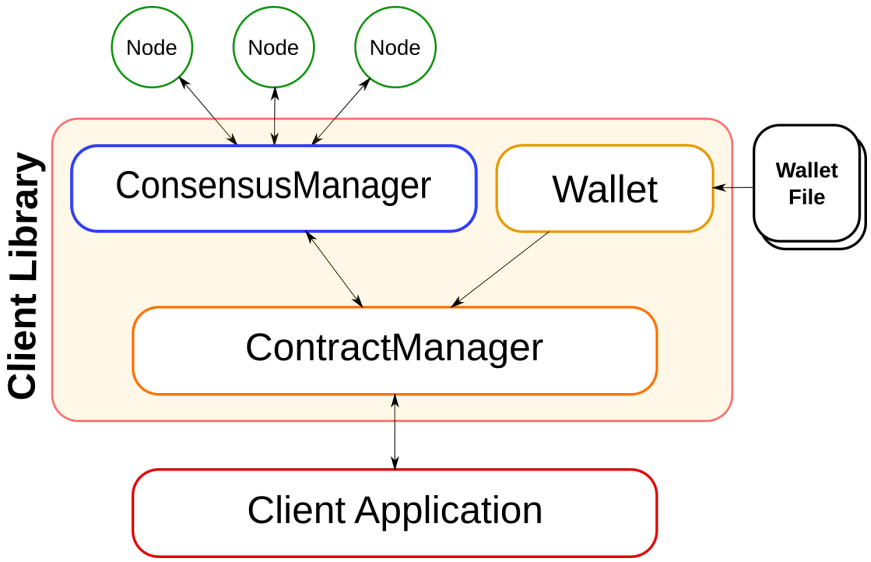

Figure 5.1: Client library modules with TST module (ContractManager)

5.1 ConsensusManager module

The *ConsensusManager* module maintains a list of active nodes and offers the abstraction to apply the majority mechanism in this list. It also provides the option to integrate a reputation system with different behaviors.

Every time that a plugin specific module needs to receive an information from the nodes, it calls the *ConsensusManager.JsonConsensusCall()* method; this call sends the information request to all available nodes, or to a part of them (depending by the policy used) and waits for replies. After that, if all replies are equal, the module accepts the value as true value; if some nodes send a different value, the chosen value is the one provided by the majority of nodes as true if the reputation system is disabled, or the one with the best score otherwise.

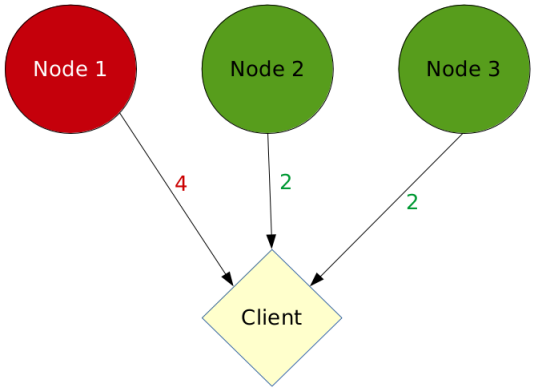

Figure 5.2: Example where a client is connected with 3 nodes; one of them sends wrong data

5.1.1 Reputation system

To protect client applications from malicious behavior of nodes, the ConsensusManager module has the possibility to integrate a reputation system that gives a different score to each connected node. There are many facts that should be analyzed before implementing a reputation system; for that purpose, we referenced to the article *A survey of attack and defense techniques for reputation systems* [17].

In this system a reputation value is assigned for each node connected to the client application; the reputation value for a node starts from a default value, and is updated deterministically: it will increase with a positive feedback and it will decrease with a negative feedback. When a reputation value for a node reaches a low value, the node is removed from the consensus group.

Another important aspect is to define how feed-backs are assigned; in the client library the only aspect that can be converted to a feedback for a node reputation, is how many times a node falls outside the consensus set when performing a *JsonConsensusCall()*: if the node falls in the majority, we have a positive feedback, a negative feedback oth-

erwise; the inclusion of positive feed-backs, introduces the self-promoting attack. Even if the possibility of this attack exists, both the behaviors (only negative feed-backs and both negative and positive feed-backs) are integrated in the prototype; however, the self-promoting attack is mitigated introducing an upper bound for the reputation value.

Since there is no evidence that a node falls or not in a majority set during a call, a secure dissemination mechanism could not be implemented easily; therefore we decided to use an asymmetric mode where feed-backs are not disseminated and where each client calculates the reputation of nodes which is connected to, by using only the feed-backs produced during the execution of the application, and these data are not disseminated. This implies that many of known attacks are not possible with this reputation system; prevented attacks are the slandering and the *Sybil attacks*. The only possible attack of the reputation system and of the consensus manager in general, is an orchestrated attack where the majority of nodes acts as malicious sending the same wrong value (more details about possible attack scenery are given in Chapter 6).

When reputation is enabled, nodes' reputation are used instead of the majority criteria; *ConsensusManager* will choose as the returned value the group of nodes that has the best reputation sum. *The ConsensusManager* implements three types of policies for the reputation system:

- Only negative feedback: the reputation value of a node can only decrease
- Both positive and negative feed-backs: the reputation value could increase or decrease
- No reputation: the reputation system is disabled

5.2 Wallet module

The *Wallet* module provides methods to handle a Bitcoin wallet. Private-public key pair can be loaded from a file or be randomly generated. After the creation/load of the pair, the module can produce and sign bitcoin transactions with a custom payload (messages). The Wallet also provides methods to retrieve spendable outputs and the balance of an account.

Chapter 6

Attack scenery

The use of the decentralized system makes some situations that are dangerous in a centralized system harmless. We found two classes of attacks; in one, the system (or a set of nodes in the decentralized system) acts as malicious, in the other, the system (or a set of nodes in the decentralized system) is unreachable for a generic reason. For each scenery, we explain how the situation is handled by a centralized system and by our decentralized system.

6.1 Malicious system

It could happen that a system acts maliciously; the clients in a centralized software expect that the server is a trusted authority and always acts as honest. This is not true; the network could be compromised, the server software could be modified with malicious code, or the client simply does not trust the authority.

6.1.1 System broadcasts modified data

In this scenery, the system broadcasts modified data; for example in the TST case study (Section 7.4) when a client tells a contract, the system can save/broadcast a different contract.

Centralized In the centralized system, the client has no way to detect if the system is broadcasting modified data.

Decentralized In the decentralized system, this problem is solved exploiting the blockchain signature mechanism; all messages are signed with the client private key. The node cannot modify the message meta-data because it has not the client private key; if it modifies message data in the DHT, other nodes will not accept the message because of an hash mismatch. If the client after an interval does not find its message in other nodes, the client automatically broadcasts the same message by using another node.

6.1.2 System deletes data received from clients

In this scenery, the system deletes data received from clients.

Centralized In the centralized system, the client has no way to detect if the system is deleting data.

Decentralized In the decentralized system, the client library periodically asks to other nodes if the message is securely saved in the block-chain. If after a timeout the message is not present, the client library tries to broadcast again using a different node.

6.1.3 System sends wrong data to clients

In this scenery, the system sends wrong data to clients.

Centralized In the centralized system, the client has no way to detect if the system is sending wrong data to clients.

Decentralized In the decentralized system, when a client needs some data, it asks to all nodes which it is connected to; through the majority mechanism, the client chooses the most shared data as valid (or the data with the best score, if the reputation system is enabled). Wrong data are accepted as valid by a client, only if the majority of connected nodes sends the same wrong value.

6.2 Unreachable system

In this scenery, the system is unreachable for various reasons: the most common ones are DOS attacks, hardware faults, malicious code injected, server overload.

Centralized In the centralized system, this scenery leads to the system unavailability; all clients cannot connect to the server. Even in short time unreachability, a side effect of this scenery is for example in the TST case study a player that become guilty in a session because he could not perform an action.

Decentralized In the decentralized system, the unavailability of a single node do not compromise the entire system because a client is usually connected to more than one node. If the number of unreachable nodes becomes greater than one, the decentralized system could be unreachable by some clients that are connected only to unreachable nodes; if this situation happens, the client can always get new available nodes.

Chapter 7

Case study plugins

To validate the general applicability of our system, we develop four case studies, which include a plugin, the corresponding client library, and some usage examples.

The first case study is a simple *HelloWorld* plugin, where clients can save their name and get previously saved names with their occurrences; the main purpose of this toy example is to present a walk-through of how a plugin and its client library is written. The second case study implements in our system a key-value decentralized database, similar to the one provided by Blockstore [9]. The third case study is a *FIFO* message oriented middleware which acts as a decentralized broker for messages; with this plugin clients can produce or consume messages from queues, in a way similar from the middleware *RabbitMQ* [6]. The last case study is more complex, and implements a decentralized contract oriented middleware based on timed session types.

7.1 Hello World plugin

The HelloWorldPlugin is a simplified example of a Plugin. This Plugin allows to send only one type of message, the *hello* message, that contains a name (a string); when a node finds a new *hello* message, the new name is saved in the node private database: if the name already exists, the occurrences of that name are updated in its database. The client library allows a client application to create *hello* messages and to retrieve information about previously sent names. Even if this plugin is useless, it explains how to compose a plugin (through a source code walk-through 7.1.1), and how different parts of a plugin interact.

7.1.1 Plugin source code walk-through

In this section the plugin source code is explained. The plugin is divided in two main parts: the first (Listings 1, 2, 3 and 4) runs in the software of each node, the second (Listing 5) runs in the client library and it is exploited to write applications for this plugin.

Node part

At the beginning, in Listing 1 we define the protocol for the new plugin, and the data structure for supported messages. At line 1-4 we define a set of constants like the unique plugin code, and the code for each type of message. Then we extend the *Message* class, defining a constructor for hello message (lines 8-13) and we override the function *toJSON()* for the serialization of a new message.

```
1  class HelloWorldProto:
2          PLUGIN_CODE = 0x05
3          METHOD_HELLO = 0x01
4          METHOD_LIST = [METHOD_HELLO]
5
6
7  class HelloWorldMessage (Message):
8          def hello (name):
9                  m = HelloWorldMessage ()
10                 m.Name = name
11                 m.PluginCode = HelloWorldProto.PLUGIN_CODE
12                 m.Method = HelloWorldProto.METHOD_HELLO
13                 return m
14
15         def toJSON (self):
16                 data = super (HelloWorldMessage, self).toJSON ()
17                 if self.Method == HelloWorldProto.METHOD_HELLO:
18                         data["name"] = self.Name
19                 else:
20                         return None
21                 return data
```

Listing 1: HelloWorld plugin source code: Messages

The next step is to write the core of our plugin, extending the class *plugin.VM* as shown in Listing 2; in this class we should define all the methods that interact with the plugin state, including query and data insertion. In our case we need only one function for querying inserted names, and another one to insert a new name in the database. For each plugin, the *PluginManager* automatically creates a new internal database for storing the plugin state.

```
class HelloWorldVM (plugin.VM):
        def __init__ (self, chain, database):
                super (HelloWorldVM, self).__init__ (chain, database)
                self.database.init ("names", {})

        def addName (self, name):
                name = name.lower ()
                names = self.database.get ("names")
                if name in names:
                        names[name] += 1
                else:
                        names[name] = 1
                self.database.set ("names", names)

        def getNames (self):
                return self.database.get ("names")
```

Listing 2: HelloWorld plugin source code: VM

To use the functionality of the plugin from an external application, the plugin should offer a set of API calls; this task is done in Listing 3 extending the *plugin.API* class, creating a dict object which contains new API calls (lines 4-10). Then we write our API methods, in this example there are only two:

- `hello (name)`: creates an hello message with the passed name, returning the output script and other broadcast information
- `getNames ()`: gets a key-value list of existing names with their frequency, invoking the *VM.getNames* method

```
class HelloWorldAPI (plugin.API):
        def __init__ (self, vm, dht, api):
                self.api = api
                rpcmethods = {}
                rpcmethods["get_names"] = { "call": self.method_get_names,
                                            "help": {"args": [], "return": {}} }
                rpcmethods["hello"] = { "call": self.method_hello,
                                        "help": {"args": ["name"], "return": {}} }
                errors = {}
                super (HelloWorldAPI, self).__init__(vm, dht, rpcmethods, errors)

        def method_get_names (self):
                return (self.vm.getNames ())

        def method_hello (self, name):
                message = HelloWorldMessage.hello (name)
                [datahash, outscript, tempid] = message.toOutputScript (self.dht)
                r = { "outscript": outscript, "datahash": datahash,
                          "tempid": tempid,
                          "fee": Protocol.estimateFee (
                                  self.vm.getChainCode (), 100 * len (name)) }
                return r
```

Listing 3: HelloWorld plugin source code: API

Finally we define the plugin in Listing 4 by extending the class *plugin.Plugin*; at lines 3-8 we bind all the previously created classes, also telling the *PluginManager* how to handle each message through the *handleMessage* method in line 10-14.

```
1  class HelloWorldPlugin (plugin.Plugin):
2          def __init__ (self, chain, db, dht, apimaster):
3                  self.VM = HelloWorldVM (chain, db)
4                  super (HelloWorldPlugin, self).__init__("HW",
5                                                  HelloWorldProto.PLUGIN_CODE,
6                                                  HelloWorldProto.METHOD_LIST,
7                                                  chain, db, dht)
8                  self.API = HelloWorldAPI (self.VM, self.DHT, apimaster)
9
10         def handleMessage (self, m):
11                 if m.Method == HelloWorldProto.METHOD_HELLO:
12                         logger.pluginfo ("Found new message %s: hello %s", m.Hash,
13                                                                 m.Data["name"])
14                         self.VM.addName (m.Data["name"])
```

Listing 4: HelloWorld plugin source code: Plugin

Library

To use this plugin as a library for client applications, we should define a library class that binds the API calls described in Listing 3 inside a library; we do this by extending the *PluginManager* (Listing 5), adding support for our new methods by binding the API calls *hw.hello* and *hw.get_ names*. The *sendName* method only creates a new message that contains the given name, and broadcasts it into the network; the other method *getNames* performs a consensus query to all nodes, and returns the result.

```
1  from libcontractvm import Wallet, ConsensusManager, PluginManager
2
3  class HelloWorldManager (PluginManager.PluginManager):
4          def __init__ (self, consensusManager, wallet = None):
5                  super (HelloWorldManager, self).__init__(consensusManager, wallet)
6
7          def sendName (self, name):
8                  cid = self._produce_transaction ('hw.hello', [name])
9                  return cid
10
11         def getNames (self):
12                 cc = self.consensusManager.jsonConsensusCall ('hw.get_names', [])
13                 return cc['result']
```

Listing 5: HelloWorld plugin library source code

7.1.2 Example usage

Listing 6 shows an hello world application of the HelloWorld plugin. At lines 3-6 the ConsensusManager is created and initialized with a static set of nodes; then, in lines 8-9, a Wallet object is created using a local instance of bitcoin-core with private keys saved in the

file *app.wallet*. At line 11 the HelloWorldManager is created using the ConsensusManager and Wallet objects created before. At line 13-14 the script asks to the user for a name and sends it to the network. Then in line 15, the list of all broadcasted names is retrieved. Finally the scripts will display a different message if the name has already been submitted or not.

```
1  from libcontractvm import *
2
3  consMan = ConsensusManager.ConsensusManager ()
4  consMan.addNode ("http://192.168.1.102:9095")
5  consMan.addNode ("http://192.168.1.105:9095")
6  consMan.addNode ("http://192.168.1.107:9095")
7
8  wallet = WalletNode.WalletNode (url="http://test:testpass@localhost:18332",
9                                  wallet_file="app.wallet")
10
11 helloworldMan = HelloWorldManager.HelloWorldManager (consMan, wallet=wallet)
12
13 yname = input ('Insert a name to greet: ')
14 helloworldMan.sendName (yname)
15 names = helloworldMan.getNames ()
16
17 if yname in names:
18         print ("Your name has already greeted in", names[yname], "other messages")
19 else:
20         print ("Oh cool, you're the first that said hello to", yname)
```

Listing 6: Example usage of the client library for HelloWorld plugin

7.2 BlockStore plugin

The next example is *BlockStorePlugin*, a plugin that implements a key-value storage similar to *BlockStore* [9]. This Plugin allows to send only one type of message, the *set* message, that contains a key-value pair; when a node finds a new *set* message, it saves the key-value pair in a private database. Clients can set new keys and retrieve already set keys that have been stored in a decentralized way.

The plugin offers two API calls:

- `set (key, value)`: set the *key* with the given *value*
- `get (key)`: returns the value assigned to the given *key*

7.2.1 Example usage

Listing 7 shows an hello world application of the BlockStore plugin. At lines 3-6 the ConsensusManager is created and initialized with a static set of nodes; then, in lines 8-9, a Wallet object is created using a local instance of bitcoin-core with private keys saved in the file *app.wallet*. At line 11 the BlockStoreManager is created using the ConsensusManager and Wallet objects created before. At line 13-15 the script asks to the user for a key-value pair and sends it to the network. Then in line 17-18, another key is asked and retrieved to the client application and displayed.

```
1  from libcontractvm import *
2
3  consMan = ConsensusManager.ConsensusManager ()
4  consMan.addNode ("http://192.168.1.102:9095")
5  consMan.addNode ("http://192.168.1.105:9095")
6  consMan.addNode ("http://192.168.1.107:9095")
7
8  wallet = WalletNode.WalletNode (url="http://test:testpass@localhost:18332",
9                                  wallet_file="app.wallet")
10
11 bsMan = BlockStoreManager.BlockStoreManager (consMan, wallet=wallet)
12
13 ykey = input ('Insert a key to set: ')
14 yvalue = input ('Insert a value to set: ')
15 bsMan.set (ykey, yvalue)
16
17 ykey = input ('Insert a key to get: ')
18 value = bsMan.get (ykey)
19 print (ykey,'=',value)
```

Listing 7: Example usage of the client library for BlockStore plugin

7.3 FIFO message queue plugin

The FIFO plugin implements a message oriented middleware which acts as a decentralized broker for messages; with this plugin clients can produce or consume messages from queues, in a way similar from the middleware *RabbitMQ* [6].

The plugin offers two API calls to publish and get messages into/from a queue:

- `publish_message (queue, body)`: append a new message in the specified message *queue*
- `get_messages (queue, last)`: get last messages from the *queue*

7.3.1 Example usage

This example is composed of two scripts; the producer script (Listing 8) produces new messages in the message queue, while the consumer script (Listing 9) consumes new messages.

Both listings show an hello world application of the FIFO plugin. At lines 3-6 the ConsensusManager is created and initialized with a static set of nodes; then, in lines 8-9, a Wallet object is created using a local instance of bitcoin-core with private keys saved in the file *app.wallet*. At line 11 the FIFOManager is created by using the ConsensusManager and Wallet objects created before.

In Listing 8 at lines 13-15, the script asks for a new message body and submits it into the queue *helloqueue*. In Listing 9 at lines 13-14 a new consumer function *consume* is defined: this function only prints new messages; at lines 16-17 *FIFOManager.consume* binds the previously defined function as the handler for new messages on *helloqueue* then it starts the consumer loop. When a new message arrives in the queue, *consume* is called with the body of the new message.

```
1  from libcontractvm import *
2
3  consMan = ConsensusManager.ConsensusManager ()
4  consMan.addNode ("http://192.168.1.102:9095")
5  consMan.addNode ("http://192.168.1.105:9095")
6  consMan.addNode ("http://192.168.1.107:9095")
7
8  wallet = WalletNode.WalletNode (url="http://test:testpass@localhost:18332",
9                                  wallet_file="app.wallet")
10
11 fifoMan = FIFOManager.FIFOManager (consMan, wallet=wallet)
12
13 body = input ('Insert a body for the message: ')
14 fifoMan.publish ('helloqueue', body)
15 print ('Done.')
```

Listing 8: Example usage of the client library for FIFO plugin: message producer

```
from libcontractvm import *

consMan = ConsensusManager.ConsensusManager ()
consMan.addNode ("http://192.168.1.102:9095")
consMan.addNode ("http://192.168.1.105:9095")
consMan.addNode ("http://192.168.1.107:9095")

wallet = WalletNode.WalletNode (url="http://test:testpass@localhost:18332",
                                wallet_file="app.wallet")

fifoMan = FIFOManager.FIFOManager (consMan, wallet=wallet)

def consume (queue, body):
        print ('[x] Received:', body)

fifoMan.consume ('helloqueue', consume)
fifoMan.startConsumer ()
```

Listing 9: Example usage of the client library for FIFO plugin: message consumer

7.4 Timed Session Types plugin

In the first part of this master thesis we showed the theoretical background required for understanding the contract model based on timed session types (Chapter 2). In this case study we propose the implementation of a Plugin that allows to write applications using this contract-oriented paradigm.

This plugin re-implement a previous work of a centralized message oriented middleware based on timed session types theory, that allows players to advertise contracts, find a compliant other contract, initiate a session and fire actions into a running session; all of these operations are monitored by the middleware.

As the centralized system, two external software applications are used: the first, *Uppaal*, is a toolbox for modeling, simulation and verification of real-time systems, based on constraint-solving and on-the-fly techniques, developed jointly by Uppsala University and Aalborg University. The Uppaal software allows to describe timed automata and to execute query on different characteristics of the automata. The second, *Tibet*, is a software project of the Department of Mathematics and Computer Science of the University of Cagliari, whose purpose is to translate contracts, expressed in abstract Ocaml syntax, into UPPAAL's automata, in order to automate the compliance checking.

Since we are using timed session type, we need a way to achieve a global time unit measurement shared between nodes; given that each block included in the block-chain is stored at a certain height, the height value of the block that hosts the message is shared identical between each node connected, and it can be exploited as a time unit measure. This implies that a time unit has an average duration that depends on the *average block time* of the specific block-chain used.

7.4.1 Messages

The plugin can handle 4 types of messages: every type of message is handled in different ways by the plugin software.

tell A *tell* message contains a contract advertised by a player. When a *tell* message arrives in a node, the contract is checked for validity; then, it is inserted in the contracts database table and its hash goes in the pending contracts database table. If the node contains contracts told by its clients, the node checks if a compliant contract exists; if so, the compliant contracts are associated with the new told contract.

fuse A *fuse* message composes two contracts in one session. When a *fuse* message arrives, the node retrieves the two interested contracts and checks if they are compliant. If so, a new session is stored in the database, contracts are removed from pending state and a new session state is stored in the database. Unlike the centralized version, in this system more than one fuse can appear in the block-chain for a given contract; if the fuse messages are in a different block, the oldest message is chosen. If they are in the same

block, the first fuse of the block is chosen as valid (the transactions in blocks have the same order in all bitcoin nodes).

accept An *accept* message creates a session where a contract is composed with its dual. This message is handled similarly to the *fuse* message.

do A *do* message contains a player action for a given session. When a *do* message arrives, the node retrieves the associated session and updates the session state, detecting if one of the involved players is violating its contract. The updated state is then saved in the database.

7.4.2 Database

Every node with the *TST* plugin enabled, maintains a private database that holds the system state; this includes all told contracts not expired or fused, the reputation value of each player, all running and ended sessions with performed actions and current session state. This information is updated only by new messages or by the time passing; this implies that all the nodes in the same network have the same information in their private databases.

7.4.3 API

The *TST* plugin implements specific TST APIs, divided in three classes; the first allows the client to create a transaction that contains a TST message, the second allows to query the system state and the third allows to perform operations on contracts:

Transaction creation

- `tell (contract_xml, player_address, expire_block_delta)`: creates a tell message
- `fuse (contract_a_hash, contract_b_hash, player_address)`: creates a fuse message
- `accept (contract_hash, player_address)`: creates an accept message
- `do (session_hash, action, value, nonce, player_address)`: creates a do message

System state query

- `listcontracts (type=all|pending|fused|ended)`: lists all contracts or a part of them depending on their types
- `listsenssions (type=all|running|ended)`: lists all sessions or a part of them depending on their types
- `getcontract (contract_hash)`: gets information about a single contract
- `compliantwithcontract(contract_hash)`: returns a list of contracts that are compliant with a given contract
- `getsession (session_hash)`: gets information about a single session
- `getaction (action_hash)`: gets information about a single action
- `getplayerreputation (player_address)`: gets the reputation of a player

Contract operations

- `validatecontract (contract_xml)`: returns true if the contract is valid
- `dualcontract (contract_xml)`: returns the dual of a contract, if exists
- `translatecontract (contract)`: translates a contract from raw to xml
- `checkcontractscompliance (contract_a_xml, contract_b_xml)`: checks compliance between two contracts

7.4.4 Client library

This plugin is also integrated in the prototype client library through *ContractManager* module.

The *ContractManager* module is the core of the client library for TST and provides the abstraction for a contract/session and related methods. The module should be initialized with a valid *ConsensusManager* and a *Wallet*, and optionally a previously told contract. After the initialization, the library can be used as the centralized version.

In contrast to the centralized middleware where the middleware server fuses compliant contracts, in the decentralized system it is the client that must fuse the session with the broadcasting of a *fuse* message: nevertheless this action is totally transparent for the application and it is handled automatically by the library. When the contract appears in the block-chain, the client library polls the node for a compliant contract: when a compliant contract is found, the client library prepares and broadcasts the transaction for the fuse.

The *ContractManager* main primitives are:

- tell: advertise a new client contract
- send: send a new client action to the current session
- receive: receive data from the current session

7.4.5 Example usage

Listing 10 shows a simple *Hello world* example where a player advertises a contract, waits for a compliant, establishes a session and follows contract rules to do its job.

At lines 3-8 the *ConsensusManager* is created and initialized with a static set of nodes; then, in lines 10-11, a *Wallet* object is created using a local instance of bitcoin-core with private keys saved in the file *app.wallet*. At line 13 the *ContractManager* for TST is created using the ConsensusManager and Wallet objects created before. At line 15 the contract of A *!greet{;t}.?planet{t<7}* is translated to an XML contract (with *translate()*), then the contract is broadcasted to the node network.

At line 16 a blocking call to *waitUntilTold* waits until the contract of A is securely stored in the block-chain. At line 17-18, the unique contract hash of A is displayed. At line 20 a blocking call to *waitUntilSessionStart* waits until the contract of A is fused with a compliant contract told by another player (say, B). At line 22, following the contract, A sends a message with label *greet* with the value *hello*; this message also resets clock *t*. If B is following the contract, B should receive the *greet* message and send back a message with

label *planet* to player A before the timer reaches 7 time units. At lines 24-28, if B follows the contract, A will receive and print the *planet* message and the session ends; otherwise, the B player becomes culpable and the session ends, raising the *SessionEndedException*.

```
1  from libcontractvm import *
2
3  consMan = ConsensusManager.ConsensusManager ()
4  consMan.addNode ("http://192.168.1.102:9095")
5  consMan.addNode ("http://192.168.1.105:9095")
6  consMan.addNode ("http://192.168.1.107:9095")
7  consMan.addNode ("http://192.168.1.109:9095")
8  consMan.addNode ("http://192.168.1.110:9095")
9
10 wallet = WalletNode.WalletNode (url="http://test:testpass@localhost:18332",
11                                 wallet_file="app.wallet")
12
13 contMan = ContractManager.ContractManager (consMan, wallet=wallet)
14
15 contMan.tell (contMan.translate ("!greet{;t}.?planet{t<7}"))
16 contMan.waitUntilTold ()
17 h = contMan.getHash ()
18 print ("Contract", cm1.getHash (), "told")
19
20 contMan.waitUntilSessionStart ()
21
22 contMan.send ("!greet", "hello")
23
24 try:
25         contMan.waitUntilReceive ()
26         print ('Hello', contMan.receive ()['planet'])
27 except (ContractException.SessionEndedException):
28         print ("The other player is culpable of violation")
```

Listing 10: Example usage of the client library for TST plugin

7.4.6 Contracts explorer

A graphical tool to explore the current state of the plugin has been developed using the client library, exploiting query related API to retrieve information. The tool's primary purpose was the debugging of the prototype, but it is also useful to check the state of running contracts; the tool is divided in 3 main views: the first view shows a summary of the system state, with the number of running sessions and pending contracts (Figure 7.1). The second view shows detailed information about a contract (Figure 7.2); the last view shows information about a session, including performed actions and session state (Figure 7.3).

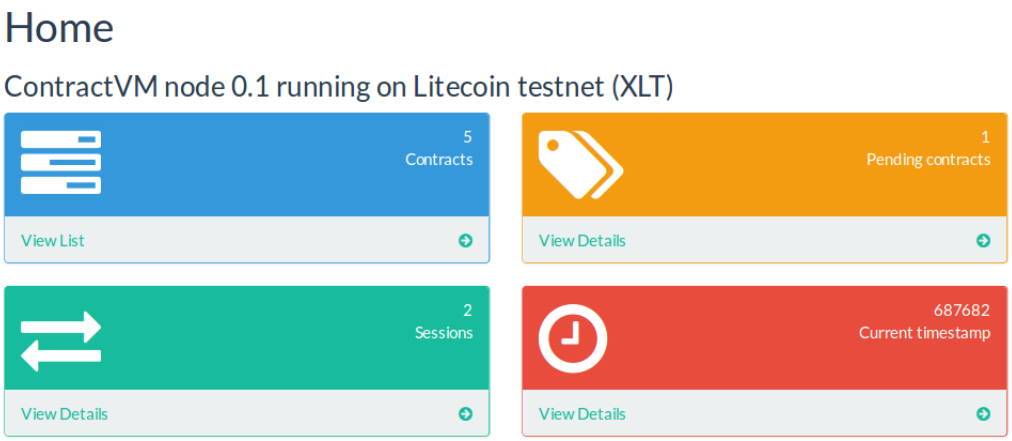

Figure 7.1: Contracts explorer: system overview

Figure 7.2: Contracts explorer: single contract

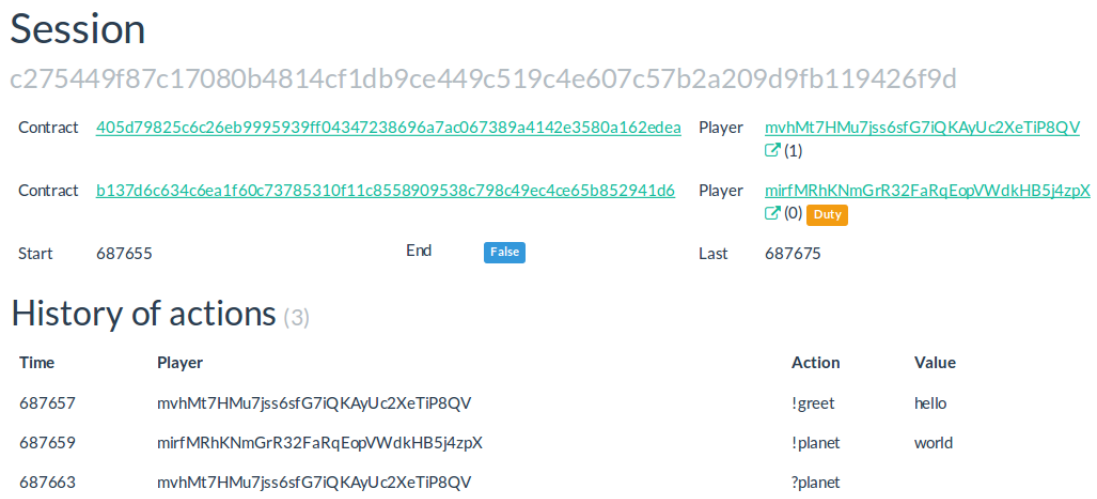

Figure 7.3: Contracts explorer: single session with trace of performed actions

Chapter 8

Conclusions and future works

The goal of this Master's thesis was to design and develop a working prototype of a decentralized block-chain based message oriented middleware and then implement the timed session type contract model. The resulting prototype, however, is a more general framework where a plugin system allows the implementation of different coexisting decentralized applications.

The steps taken to achieve this goal were preceded by the study of the previous works about decentralization and block-chain, of the contract model of timed session types and of the existing implementation of the centralized contract-oriented middleware based on timed session types, written by Livio Pompianu and Sebastian Podda.

Then, we explored different existing software and libraries for writing decentralized applications; the best choice at the moment of the thesis was to use the existing block-chain of Bitcoin or one of its forks. After this, the design part started then coming to a working *python3* prototype. Following this, we developed four different plugins for the prototype; the most relevant is the TST case study 7.4 that exploits the timed session types theory to create a working contract-oriented middleware.

8.1 Source code

All the produced software, including the node daemon with all case studies [13], the client library [14] and the Kademlia DHT [15], are available as open-source projects on *Github* [4], so every one can download the software, deploy a node or run client applications.

8.2 Related works

There are several works, in literature, related to that carried out in this thesis. In this section we explore a list of some relevant decentralized software with their features and issues.

Bitcoin [19] is a decentralized peer-to-peer electronic cash system. Bitcoin proposes a financial system where all transactions are stored in a public ledger, called *block-chain*;

the block-chain is updated with new blocks of transactions, using a distributed consensus process called *mining*, that enforces a chronological order in the block-chain, protects the neutrality of the network, and allows different computers to agree on the state of the system. To be confirmed, transactions must be packed in a block that fits cryptographic rules, verified by the network. These rules prevent previous blocks from being modified since a change would invalidate all following blocks. Mining also creates the equivalent of a competitive lottery that prevents any individual from easily adding new blocks consecutively in the block chain. This way, no one can control what is included in the block-chain, or replace parts of the block chain to roll back their own spends.

Metadisk [20] is a decentralized cloud storage (similar to centralized services like *GoogleDrive* [5] or *Dropbox* [1]) built on top of an existing *bitcoin* based block-chain. In this system, users can rent their bandwidth and disk space to other users that need cloud storage. As *blockstore*, *metadisk* uses the block-chain to store files meta-data while uses an external DHT for data storage. The main difference between *metadisk* and *counterparty*, is that *metadisk* is designed as an end-user application for cloud storage, and it does not implement the possibility of other usages.

CounterParty [18] is a decentralized application for creating peer-to-peer financial products on the Bitcoin block-chain; its protocol is primarily designed to create virtual assets and issue dividends. The CounterParty protocol defines a set of rules for embedding messages inside block-chain transactions; it also defines a set of default message types used to perform financial operations on assets (send assets, create dividends, publish order and few others). *counterpartyd* was the first reference implementation of the protocol, and consists in a python software that runs on top of the bitcoin-core node software; counterpartyd nodes exchange data only using the bitcoin block-chain, and this is possible because message are completely embedded in bitcoin transactions. Even if this seems to be a good approach, two main criticisms arise about this technique: the first regards the block-chain flooding problem because each CounterParty message is downloaded by each bitcoin node. The second regards the limit of the size of messages that could be included in the block-chain; even if the limit is increased using some block-chain tricks (message data is encrypted in multi-signature transaction despite this is not the goal of a multi-signature transaction), the limit is still too low for many applications.

8.3 Future works

There are many other improvements that are necessary for a production usage of this software and many other ways to do things that need to be explored.

The first improvement to the existing prototype, is to write a more versatile client library; it can be rewritten in a more low level language (like e.g. C++ or C) in the aim to make it usable from different environments (using for example *SWIG* [7], a software development tool that connects programs written in C and C++ with a variety of high-level programming languages).

In our system, users are not encouraged to deploy and run a node; the incentive is necessary because a node offers processing, storage and network resources. A possible solutions is to offer a reward for data storage using an approach similar to the one used in Metadisk [20] where a decentralized algorithm checks how many messages are stored in a node, and the system pays it for these resources. Another possibility is to give a reward to a node for each broadcasted message, taking a part of the fees.

To avoid the need of a running instance of bitcoin-core, the prototype could implement the bitcoin protocol part related to the block retrieval. With this modification, all transaction data that do not belong to plugin messages will not be stored in the node machine. A node bootstrapping mechanism is also necessary for the automation of node probing both for node and client library.

Another useful future work is the implementation of a package manager for installing and updating available plugins.

A possible future research work is to re-implement the timed session types plugin with an Ethereum dapp; the challenging part of this new approach is to implement the model checker and the execution monitor using one of the Ethereum language. Another challenge is to write a small dapp, because to run a software in the Ethereum network you should pay a fee for each process depending on the number of operations performed. The first advantage of this approach is that nodes should not install a third party software, but they only need the Ethereum official client; the second advantage is that the reward for nodes is already integrated in the Ethereum mining process. After the deploy on the network, the dapp will stay alive until users continue to use it.

List of Figures

List of listings

Bibliography

[1] Dropbox. https://dropbox.com.

[2] bittorrent. http://www.bittorrent.com.

[3] emule. http://www.emule-project.net.

[4] Github. https://github.com.

[5] Google drive. https://drive.google.com.

[6] Rabbitmq. https://www.rabbitmq.com.

[7] Swig. http://www.swig.org.

[8] Namecoin: a decentralized dns service. https://wiki.namecoin.org/, 2011.

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

[10] Massimo Bartoletti, Tiziana Cimoli, Maurizio Murgia, Alessandro Sebastian Podda, and Livio Pompianu. Compliance and subtyping in timed session types. In *FORTE 2015*, pages 161–177, 2015. doi: 10.1007/978-3-319-19195-9_11.

[11] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *ACPN*, pages 87–124, 2003. doi: 10.1007/978-3-540-27755-2_3.

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

[13] Davide Gessa. Contractvm daemon source code. https://github.com/contractvm/contractvmd, .

[14] Davide Gessa. Contractvm library source code. https://github.com/contractvm/libcontractvm, .

[15] Davide Gessa. Kad.py kademlia dht source code. https://github.com/contractvm/kad.py, .

[16] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Inf. Comput.*, 111(2):193–244, 1994.

[17] Kevin Hoffman, David Zage, and Cristina Nita-Rotaru. A survey of attack and defense techniques for reputation systems. *ACM Computing Surveys (CSUR)*, 42(1): 1, 2009.

[18] Ouziel Slama Robby Dermody, Adam Krellenstein and Evan Wagner. Counterparty: Protocol specification. http://counterparty.io/docs/protocol_specification/, 2014.

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

[20] Jim Lowry Shawn Wilkinson and Tome Boshevski. Metadisk: a blockchain-based decentralized file storage application. http://metadisk.org/metadisk.pdf, 2014.