ETSI GR PDL 004 V1.1.1 (2021-02)

Permissioned Distributed Ledgers (PDL)
Smart Contracts
System Architecture and Functional Specification

Disclaimer

The present document has been produced and approved by the Permissioned Distributed Ledger ETSI Industry Specification Group (ISG) and represents the views of those members who participated in this ISG.
It does not necessarily represent the views of the entire ETSI membership.

Reference

DGR/PDL-004_smart contract

Keywords

blockchain, policies, PDL, SLA, smart contract

ETSI

650 Route des Lucioles
F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00   Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C
Association à but non lucratif enregistrée à la
Sous-Préfecture de Grasse (06) N° 7803/88

Contents

Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The information pertaining to these essential IPRs, if any, is publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (https://ipr.etsi.org/).

Pursuant to the ETSI IPR Policy, no investigation, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

Foreword

This Group Report (GR) has been produced by ETSI Industry Specification Group (ISG) Permissioned Distributed Ledger (PDL).

Modal verbs terminology

In the present document "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the ETSI Drafting Rules (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

Executive summary

The present document specifies a high-level functional abstraction of PDL Smart Contract System Architecture. In particular, basic building blocks for designing, coding and testing Smart Contracts for the PDLs. This includes describing how different classes of systems interact with Smart Contracts. Processes, models, and detailed information are beyond the scope of the present document.

Introduction

The present document defines a high-level functional abstraction of policies to design and code Smart Contract components. Smart Contracts are mere codes, and if not well planned, designed, coded and tested can leave the system vulnerable to external attacks and internal errors.

1 Scope

The present document specifies the functional components of Smart Contracts, their planning, coding and testing. This includes:

a) reference architecture of the technology enabling Smart Contracts - the planning, designing and programming frameworks;

b) specify how to engage using this architecture - the methods and frameworks the Smart Contracts building blocks possibly communicate;

c) point out possible threats and limitations.

2 References

2.1 Normative references

Normative references are not applicable in the present document.

2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] ACM Digital Library: "Securify: Practical Security Analysis of Smart Contracts".

NOTE: Available at https://dl.acm.org/doi/pdf/10.1145/3243734.3243780.

[i.2] ACM Digital Library: "SmartCheck: Static Analysis of Ethereum Smart Contracts".

NOTE: Available at https://dl.acm.org/doi/pdf/10.1145/3194113.3194115.

[i.3] ITU-T Report: "Distributed Ledger Technologies and Financial inclusion".

NOTE: Available at https://www.itu.int/en/ITU-T/focusgroups/dfs/Documents/201703/ITU_FGDFS_Report-on-DLT-and-Financial-Inclusion.pdf.

[i.4] ETSI GR PDL 003: "Permissioned Distributed Ledger (PDL); Application Scenarios".

NOTE: Available at https://portal.etsi.org/webapp/WorkProgram/Report_WorkItem.asp?WKI_ID=57511

[i.5] United Nations Commission on International Trade Law.

NOTE: Available at https://uncitral.un.org/

[i.6] Decentralized Public Key Infrastructure.

NOTE: Available at https://medium.com/hackergirl/decentralized-public-key-infrastructure-4e7ea9173bac

3 Definition of terms, symbols and abbreviations

3.1 Terms

For the purposes of the present document, the following terms apply:

coin: implementation using a unique ledger and usually used for financial transactions (e.g. Ether, Bitcoin)

eternal contracts: contracts which are active for infinite time

mainnet: ledger in-production

NOTE: The contracts and transactions on a mainnet are ultimate.

master-chain: primary chain where the executions of the Smart Contract are recorded

off-chain smart contract: smart contracts stored away from the ledger (i.e. trusted database or side-chain) and their execution may depend on on-chain contracts (i.e. on-chain contract can initiate off-chain contracts) and later the state can be updated

on-chain smart contract: contract that resides in the master-chain and on side-chain, that is executed directly without the instantiation of any other contract

NOTE: The beneficiaries get rewarded as soon as the contract is executed without the involvement of any other contract.

participants: participants are the members of the PDL which keep the copy of the ledger and take part in the consensus

Ricardian contract: single contract document which is both easily readable by human and machines and not self-executable

NOTE 1: It is formatted as a text file and digitally signed by the issuer of the contract.

NOTE 2: The security of a Ricardian contract is achieved by OpenPGP and all the signing keys are included within the contract so eliminates the use of external certificate authority, in other words a Ricardian contract carries its own PKI with them.

NOTE 3: The Difference between Ricardian contract and Smart Contract: The major difference between the Smart Contract and the Ricardian contracts is that Smart Contracts are executable code but Ricardian contracts are the agreements recorded in a single file and not executable on their own. A Smart Contract does not have to be a Ricardian contract and a Ricardian contract is not a Smart Contract, but a Smart Contract can execute a Ricardian contract.

side-chain: chain(s) which work as a secondary chain to the main-chain/ledger

NOTE: It can be used to off-load some of the computations for scalability or privacy.

Smart Contract (SC): computer program stored in a distributed ledger system, wherein the outcome of any execution of the program is recorded on the distributed ledger

NOTE: A Smart Contract might represent terms in a contract in law and create a legally enforceable obligation under the legislation of an applicable jurisdiction. A Smart Contract may but does not have to be human readable and is self-executable. Any executable code stored on a PDL is dubbed a "Smart Contract" (SC).

The focus of the present document is Smart Contract as software codes and is different from legal contracts.

stakeholders: parties that benefit from the PDL

NOTE: All the stakeholders may or may not keep the copy of the ledger (i.e. act as a node) and take part in consensus.

testnet: ledger or sandbox on which Smart Contracts can be installed to test their working and performance prior to installation on a mainnet

NOTE: Testnets are installed to test the performance of the code and the transactions and Smart Contracts are for the testing purposes only.

Table 3-1: Comparison of Ricardian and Smart Contract

Contract Type	Machine-Readable	Human-Readable	Self-Executable
Ricardian Contract	Yes	Yes	No
Smart Contract	Yes	Optional	Yes

3.2 Symbols

Void.

3.3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

API	Application Programming Interface
DLT	Distributed Ledger Technology
ETSI	European Telecommunications Standards Institute
GDPR	General Data Protection Regulation
ICT	Information and Communications Technology
ITU	International Telecommunication Unit
PDL	Permissioned Distributed Ledger
QoS	Quality of Service
SC	Smart Contract
SLA	Service Level Agreement
TEE	Trusted Execution Environment
UNCITRAL	United Nations Commission on International Trade Law

4 Introduction to Smart Contracts

4.1 Introduction

A Smart Contract is a computer program deployed on a PDL. The primary purpose of smart contract to keep certain software in a PDL that execute on certain execution requests.

Any PDL's general goal is the distributed management of a common data repository defining a current global state; there is no assumption on the type of data stored. When such data is an executable code (i.e. smart contracts), the induced global state can be seen as the state of a distributed virtual machine.

4.2 Object-Oriented Paradigm

Historically, the main model adopted for SCs has been along the line of the traditional Object Oriented paradigm. As such, a SC is seen as a code entity composed of two main clauses:

- Internal storage, in the form of identifiers - value associations akin to a dictionary, similarly to object fields.
- Functions' definitions, specify the set of actions allowed for the given SC with the appropriate scope modifiers, similarly to object methods.

Similar to the concepts of Object-Oriented programming a Smart Contract is instantiated from a class, and once instantiated holds a unique identifier; that is to say every instantiation is unique. The deployed Smart Contract holds a global state which means that all its fields and functions become visible and callable by other contracts (depending on access rights). Moreover, a deployed Smart Contract can be called as many times as required; however, this is dependent on the implementation.

Smart contracts have different implementations depending on the technology and consensus mechanism such as PDL types (e.g. Hyperledger, Quorum)

4.3 Properties of Smart Contracts

4.3.1 Introduction

The properties of Smart Contracts directly depend on the properties of the underlying PDL and some properties due to their requirements.

4.3.2 Immutability

As any data on a PDL, an SC is immutable; this means that a Smart Contract code, once accepted through consensus, cannot be changed. However, modifications through other methods such as proxy contracts or introducing a new Smart Contract, are possible. In such an event, the old version of the contract remains in the chain. A consequence of immutability is Importability which means that it cannot be deleted from the ledger after deployment. This brings the challenges of scalability as a PDL might be populated with dormant contracts over time. The details on scalability are discussed in later clauses.

The values contained inside an SC's internal storage are mutable as expected through function calls; for example, in an auction contract bid values will change with new bids but the final winning bid may be immutable.

4.3.3 Availability

In the case of on-chain SC, it is always available as long as the underlying master ledger is accessible. This means that a SC function can be invoked, and its fields (i.e. variables) can be read, by an entity as long as the entity has the appropriate privileges specified by the contract and the PDL. However, in the case of off-chain Smart Contracts, if the ledger where the contract is installed (i.e. secondary PDL) is not available, the SC is not accessible by the master PDL.

4.3.4 Transparency

Any entity, with the appropriate privileges, might inspect a SC code and current values. As such, it is transparent to all intended participants of the PDL. Transparency is not to be confused with immutability; a contract code remains unchangeable even though it is transparent to both parties.

Moreover, any call to a function of a contract is performed through a general state update on the PDL (i.e. transaction). As such, all function calls are recorded in the PDL and traceable by the members of the PDL with appropriate access rights.

4.3.5 Self-Execution

Any execution of a SC, i.e. an invocation to one of its visible functions, is performed by the PDL nodes, not by the user invoking the SC, nor by the SC creator. The SC execution is protected by the distributed consensus of the PDL; as such, it is beyond the control of any single party to execute a Smart Contract without the approval of PDL members. This property induces the sub-properties of:

- Atomicity: an SC invocation runs entirely or fails without affecting the state (i.e. there is no such thing as partial SC execution).
- Synchronicity: an SC invocation is executed in a synchronous way (i.e. every member with appropriate access rights get the update).
- Determinism: an SC invocation returns the same result for any node executing it.

4.3.6 Reusability

SCs are coded once and can be executed multiple times depending on PDL governance. A given Smart Contract can be used as a template for a wider set of applications sharing the same high-level logic. The actual behaviour of a given contract may change depending on the parameters which are set at invocation time. For example, the SC for cellular service is modelled with required fields for QoS metrics such as latency; all the telecom operators, in this case, will be required to specify the latency as a parameter.

4.4 Storage

Smart Contracts are typically stored in distributed ledgers; however, their storage depends upon the nature of the ledger architecture. For example, in case of a permissionless blockchain such as Ethereum, a Smart Contract will be stored by all nodes; on the contrary, in a permissioned blockchain such as Hyperledger, Smart Contracts are stored only on the nodes that are part of a given channel (an abstract point-to-point link between nodes) and are established through communication between nodes.

For off-chain SCs, the contracts may be stored on a trusted data storage, away from the ledger. This type of storage mechanism needs special security measures set out by the governance of the PDL.

Reusability techniques such as template contracts can be used to allow efficient storage of the contracts. The decision of storage is dependent on the implementation of a PDL, and the technology the companies adopt. The limitations due to the external existence of a contract is discussed in clause 8.

4.5 The Lifecycle of a Smart Contract

A Smart Contract is a computer program; the difference is that the Smart Contracts are immutable, so it requires great care to program them and is good be tested on several levels before deployment. This clause presents the recommended lifecycle, a Smart Contract may follow in order to avoid the dangers such as erroneous code. This recommended lifecycle consists of three phases: planning phase, coding & testing phase and deployment & execution phase. The phases are explained in detail in clause 5.

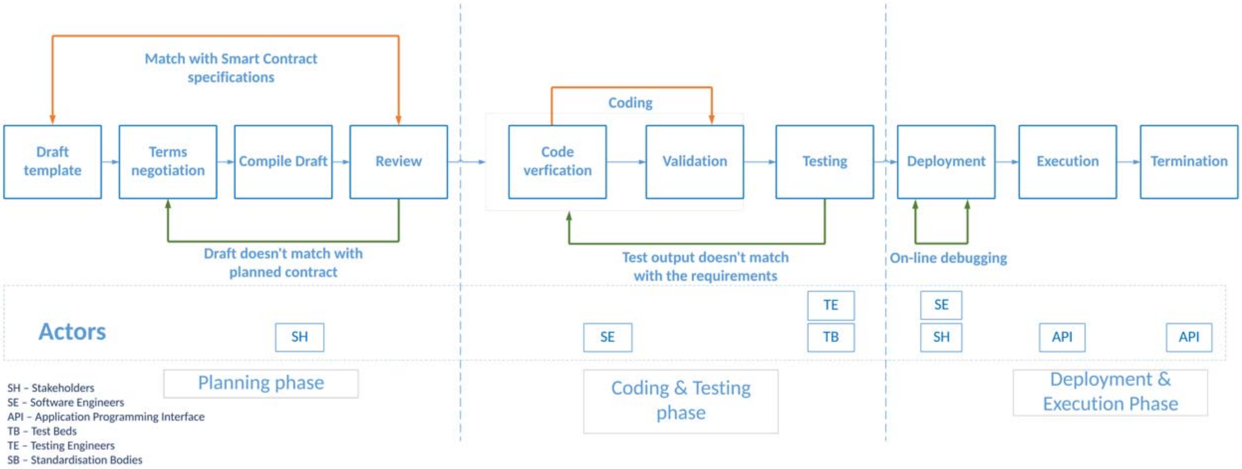

Figure 4-1: Lifecycle of a Smart Contract

5 Smart Contracts - Lifecycle phases

5.1 Introduction

Smart Contracts are software codes similar to any other software program. The difference from usual software is in the way the bugs are being fixed. The nature of PDL, does notallow backward modification of information or code so any change to Smart Contract can only be applied to the time of deployment and onwards.

Hence, careful planning and scrutiny of the code before deployment to the ledger is of utmost importance. In this clause, the stages of the Smart Contract lifecycle (Figure 4-1) are defined, which the industries may follow to implement Smart Contracts in the adopted PDL.

5.2 Planning Phase

5.2.1 Introduction

A Smart Contract can be deployed in many ways, and the deployment methods are dependent on the underlying ledger technology and acceptable by the participants through consensus. The goal is to create a contract that can be trusted by participants who do not trust each other. The planning of a Smart Contract will enable the participants to define their requirements and functionalities of a Smart Contract. The planning phase may include:

1) governance - ownership and access rights;

2) design - coding and testing;

3) deployment; and

4) management planning.

5.2.2 Governance

5.2.2.1 Introduction

A Smart Contract may define a contract, and its associated terms and conditions covering the full lifecycle of the contract, between the participants. Governance planning defines the authority of different stakeholders over the contract, for example, ownership and access rights.

Usually, the creator of a contract is the owner as well; the owner of the contracts has exclusive privileges such as contract destruction. However, in PDLs where contracts can be reused by several participants for several unrelated transactions, it is feasible to have a role-based ownership mechanism. In Role-Based ownership, the operations of a contract are governed by a group of participants with appropriate privileges; as PDL is a collaborative ledger, these privileges can be specific to a contract.

5.2.2.2 Single-party Governance

The Smart Contract, when deployed, is usually identified as being governed by a specific part (N=1) or a group of distinct parties depending on consensus and governance model.

This agreement needs to take into account the legal and business aspects of the Smart Contract, and address issues such as who is eligible to stop, terminate, or upgrade the Smart Contract, and how these are enforced contractually or technically.

Smart Contracts are a digital model of such contracts, and actors and their arrangements are beyond the scope of the present document.

5.2.2.3 Multi-party Governance

A Smart Contract may be developed for N-M interaction, i.e. one-to-one, one-to-many, many-to-one or many-to-many interactions. For example, if the contract is governed by more than one party, a consortium agreement needs to be formulated within that group to outline the governance model that is applied to the Smart Contract. Moreover, a contract may be managed by a third-party such as some stakeholders which are not directly involved in the contract.

For multi-party governance, this requires decisions on the technical implementation aspects of:

- **From whom will Smart Contracts accept the operational decisions, and how?** Since in this scenario, a Smart Contract is governed by multiple stakeholders, it is likely that some of the authorized parties/stakeholders may disagree with some decisions such as termination of a contract. In such cases, multi-signatures and voting mechanisms can be used to approve/reject a transaction.
    - In multi-signatures, group members sign a decision that is communicated to a Smart Contract and verified.
    - Another option is to use voting, in which case action is initiated, but the Smart Contract requires different parties to individually endorse the action (or reject it) within a time limit.
- **How are the governing parties recognized by the Smart Contract?** Depending on the ledger, this may be an organizational identity within the ledger, or an account owned by the party (e.g. a public key).
- **What are rights each governing entity has?** It is possible that some ledgers do not allow some actions, such as contract stop and resume, termination, contract upgrade, changes in governing party identities, and any other business-specific actions.
- **How are Smart Contracts upgraded?** If the Smart Contract can be upgraded, either via the ledger's native support (e.g. in Hyperledger Fabric, using versioned chain code), or via development techniques (e.g. proxy contract), the process of upgrade needs to be managed. This may need communication with the users of the Smart Contract as with any software release management process. If the Smart Contract is governed by a group, it is important that the group coordinate for the upgrade using the appropriate technical means.

5.2.3 Design Planning - Coding and Testing

In this stage, the stakeholders may list the coding and testing strategies and resources, they may require for later stages of coding and testing. The strategies and resources may include the following:

- Choice of programming languages.
- Choice of testing environment.
- Resources required for coding and testing such as developers and development tools.

The coding and testing phase detailed in clause 5.3.

5.2.4 Deployment Planning

5.2.4.1 Introduction

Smart Contracts can be deployed on the master-chain, side-chain or off-chain depending on the planning and requirements of the organizations. For example, if two companies are willing to run a business contract that may stay exclusively between them, they can have a side-chain with Smart Contracts deployed there and make appropriate selective updates to the main-chain such as contract start and termination dates without the details of the contract.

Table 5-1: Explanation of master-chain, side-chain and off-chain Smart Contracts

Consideration	Master-chain	Side-chain	Off-chain
Contract-type	Contract, Address of Contracts on Side-Chain and/or Off-Chain	Contract	Contract
Scalability	Limited	Limited	High
Security	High	Limited	Requires off-chain security measures
Immutability	High	Ledger-dependent	Limited
Eternity	High	Ledger-dependent	Limited
Risks	Low	Medium	High
Storage-requirement	Local	Can be distributed	Does not need to be shared
Speed	Medium	Slower	Faster
Dependency	None	Ledger and governance dependent	Ledger and governance dependent
Parallelization	Ledger-dependent	Ledger-dependent	Governance dependent

In the following clauses, the possible methods of deployment are discussed.

5.2.4.2 On-chain deployment

This is the simplest method for deployment of Smart Contracts and the contracts are stored directly in the ledger, which can be a master-chain, a side-chain or an off-chain. The advantage is that the customers do not have to rely on any other side-chain or off-chain (which may require additional resources) and it is best for a system managed by a single entity. Since all the full contract codes are stored in a single chain, in long-term scalability can be a problem.

The simplest deployment model is where the Smart Contract is never terminated. In some ledgers, a Smart Contract can always be removed, while in other ledgers this decision can be built into the Smart Contract at development (i.e. self-destructible clauses) or deployment time (i.e. by choosing to include or omit a "termination" mechanism self-destructible clause are discussed).

5.2.4.3 Side-chain deployment

In this method, the main logic of a contract is stored in a side-chain and only some indication of that contract (such as hash or address) is stored in the master-chain. The advantage of this technique is that, since it is not required for a full-contract code to be in the master-chain, this technique is scalable.

Additionally, the side-chain contract address in the master-chain can be updated by the owner of the contract through a transaction with no additional means. The danger in this type of deployment is that, if the side-chain contract is not self-destructive, it can stay forever and can be callable by other contracts, also as it is in the chain (no matter if the chain is side-chain) it occupies storage.

Side-chain Smart Contracts can be reused by other users of the PDL (delegated by the owner of the contract).

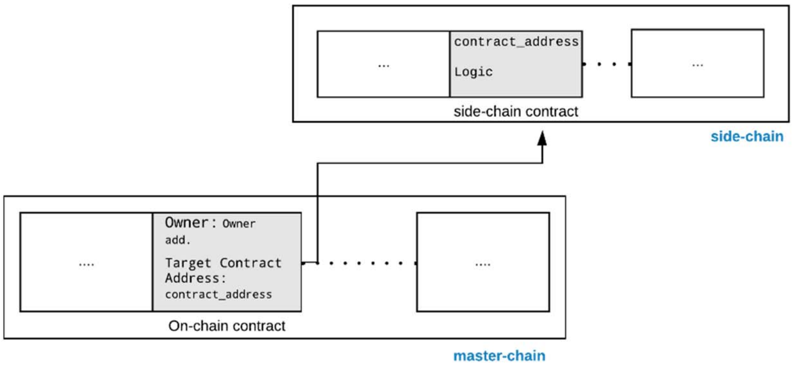

Figure 5-1: Master-chain and side-chain Smart Contracts

5.2.4.4 Off-chain deployment

In off-chain deployment, Smart Contracts are stored away from the ledger and may be in a trusted data structure. The indication of the presence of contracts such as invocations are only recorded in the master-chain or a side-chain. Off-chain deployment possess risk of trust and rely on security of the database where the contracts are stored. The major advantage of an off-chain deployment is this technique is scalable since only the invocations are stored in the PDL. Since off-chain deployment does not depend on any specific PDL, such contracts can be ported to other PDL types with relative simplicity.

5.2.4.5 Immutable deployment

There are methods by which Smart Contracts' immutability can be managed. This is typically done at the deployment stage. Some of immutability management techniques may be available natively in a specific ledger, and for other ledgers, this may require the use of programming techniques such as call delegation across contracts. Immutability as a property is discussed in clause 4.3.2.

If the ledger has immutable Smart Contracts, this governance model is recommended to be encoded within the Smart Contract during the contract planning. This is intended to stop later changes.

5.2.4.6 Terminable deployment

A Smart Contract may be terminated, i.e. permanently disabled, if the ledger or the Smart Contract itself directly supports this mechanism. A PDL is typically immutable so that Smart Contracts, but some ledgers may allow the contracts to be terminated and is dependent on the governance and the consensus of the under-lying ledger. Details of the multi-party contract is discussed in ETSI GR PDL 003 [i.4].

5.2.4.7 Upgradeable deployment

Some ledger technologies support upgrades to an existing Smart Contract, i.e. changing the Smart Contract's operational code. This typically happens by installing a master contract with a mutable field similar to passing an argument to a function. This argument acts as a pointer to another contract which carries the actual operational code. This type of deployment is useful when upgrades of a contract are needed. However, in this case, the problem of scalability exists because the old contracts may not be deleted and stay in the ledger as a dormant contract.

If the Smart Contract can be upgraded, either via the ledger's native support (such as in Hyperledger Fabric, using versioned chain code), or via development techniques (proxy contract), the process of upgrades needs to be managed. This may require communication with the users of the Smart Contract as with any software release management process. If the Smart Contract is governed by a group, the group may coordinate the upgrade using the appropriate technical means.

5.2.5 Draft template

5.2.5.1 Introduction

In the future, the PDL technology is envisioned to be used widely for all kinds of business transactions. Therefore, before the planning and coding process begins, a Smart Contract can be drafted electronically or manually. At this initial stage, some or all of the stakeholders can decide together with their requirements such as code and resources requirements. This step facilitates the smooth and error-free coding of a contract.

5.2.5.2 Terms negotiation

Once the draft of requirements is ready, the terms and conditions between the stakeholders can be decided and agreed. It is particularly important in a Smart Contracts because in traditional manual contracts, there may be a freedom of amendment at any time, whereas Smart Contract by-design do not typically have such freedom. At the same time, it is important that all stakeholders agree on terms of the entire deliberation so that there is no conflict in the future.

The terms and conditions will be varied from organization and its governance, but questions such as deployment management and lifecycle of a Smart Contract can be addressed.

Some of the important points that may be a part of the negotiation of the terms are:

1) Is the Smart Contract going to on-chain or off-chain?

2) If participants want to maintain a side-chain, who will be participants and their role?

3) For how long the side-chain will be active?

Especially in situations where contract can be stopped and resumed, terminated, or upgraded, the multi-party governance agreement may take into account who has the authority to issue these operations.

Depending on the capabilities of the ledger itself some of these policy decisions may be part of the ledger itself; in other cases, these decisions may be encoded into the Smart Contract and defined in design phase already.

5.2.5.3 Map draft template to the machine-readable context (Compile Draft)

This step provides the bridge between the draft template and the coding phase and involves the procedures in mapping the draft contract (from draft template clause 5.2.5) to a Smart Contract which is the technical representation of the same. This step not to be confused with "Compile" in the context of programming and only harmonises the template and coding steps.

This step can specify the complete supervisory level specifications such as underlying ledger technology to be used and the stakeholder needs.

5.2.5.4 Draft review (reference checklist)

The last step of the planning phase to review and verify the complete planning phase. The reference checklist may include:

1) All the stakeholder requirements are listed in the draft.

2) The planned hardware and software resources such as PDL are acceptable and reachable to all of the future nodes (i.e. participants).

3) All the functions are mapped accurately to the requirements.

4) The governance of a contract is clearly documented and part of the draft template (clause 5.2.5).

5) The contract is planned in accordance with the standardization body guidelines.

5.3 Coding and testing phase

5.3.1 Introduction

As soon as the contract plan is in place, the next step is to code it. This clause will cover the coding and testing phase of a Smart Contract and discuss the steps which can help industries produce viable contracts.

5.3.2 Coding process

The coding language decision is dependent on the underlying PDL type. Some of the PDLs may allow different languages for the Smart Contract coding, but some are very specific to this. Where there is freedom provided by a PDL type to use multiple languages, the widely used language may be adopted as they are better understood by the programmers and may have more tools available for testing and bug fixing.

5.3.3 Testing process

A Smart Contract may go through a comprehensive testing process to avoid erroneous contracts being deployed. Several steps can be part of this process, depending on the priority of organizations. A recommended testing flow is shown in Figure 5-2.

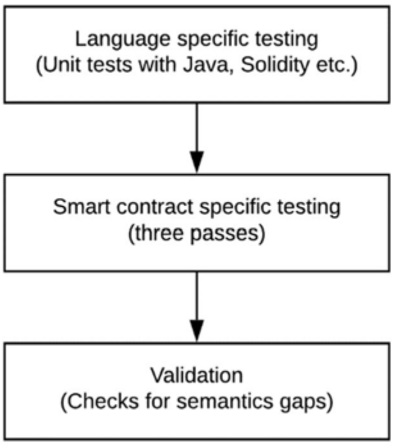
Language specific testing
(Unit tests with Java, Solidity etc.)
Smart contract specific testing
(three passes)
Validation
(Checks for semantics gaps)

Figure 5-2: Smart Contract testing process

5.3.4 Code/Programming language level Testing

The Smart Contracts' testing varies from traditional software testing in several ways.

Traditional software mostly has freedom of revision. When needed, it goes through regular updates, software revisions and patches to remove bugs. Smart Contracts are deployed on PDLs; this means all the nodes carry a copy of the same contract and execute as required. Also, any syntax or logical error will be replicated to all of the nodes, and it is impossible to fix such errors once the contract is deployed in the PDL. Such errors can be avoided by traditional language-specific software testing mechanisms. Programmers can ensure that a contract is error-free and carry out all necessary tests in a test environment before deployment.

5.3.5 Smart Contract specific testing

Smart Contract testing is different from code-level testing, as this level of testing ensures a safe and manageable Smart Contract. Smart Contracts are typically auto-executable, and their termination is difficult; hence it is important to consider following while testing a Smart Contract.

5.3.5.1 Open source SC analysers

A number of open-source SC analysers such as Securify [i.1] and SmartCheck [i.2] are available to analyse the SC code and tag the vulnerabilities present in the program. These vulnerabilities, such as uninitiated functions can provide third-party (possibly malicious) access to a contract, thus to the ledger. These analysers prevent external accesses by inspecting the code and flagging the possible vulnerabilities in the code. However, all the analysers have their limitations such as they support certain ledger technology or programming language. Also, the attacks on the contracts are evolving; hence more comprehensive scrutiny of the contracts can be achieved by multiple analysis techniques. Another important consideration for an analyser is the support for a PDL type, most of the available analysers are for Ethereum and Hyperledger and the adopters of the other ledger types can look for their respective PDL supported analyser.

5.3.5.2 Sandbox testing

A PDL is a group of nodes, and erroneous Smart Contracts can be harmful to all of the nodes. A Sandbox testing mechanism is useful before the execution of a Smart Contract on an in-production PDL to ensure safe and error-free contracts. Sandboxes are specific to the ledger type and can be local or distributed.

Local Sandbox:

- A local copy of ledger can be used as a sandbox, and sample contracts can run several times to verify the output. A disadvantage of local testing is that it may not give realistic latencies for execution and deployment. A solution for this can be a distributed full-scale Test-net.

Distributed Sandbox/Test-net:

- A solution for limitations of local sandbox can be a permanent a sandbox between the nodes or a Test-net, which serve as the testing ground only and all the Smart Contracts deployed there may not be considered as valid; to enable scalability in such sandboxes, they can be deleted after a certain time to free storage.

5.3.5.3 Three passes

It is recommended that nodes run their pre-tests before sending the deployment transaction. These pre-tests are specific to the use-case and the PDL type. For example, in a token contract, the address of the payee is important to be included in the contract, and for the asset trail contract, the change of ownership is an important parameter. Here, three reference passes for a contract are highlighted, stakeholders may look for, before deployment of their Smart Contract:

- Execution clauses:

  A contract is executed with certain predefined conditions which can be internal such as start time or external such as an API call. Hence, it is important to have the execution clauses in a contract clearly defined, as its absence will make the contract dormant. Moreover, the presence of unintended conditions can open backdoors in a contract and to be avoided.

- Penetrable clauses:

  The clauses that invoke the critical parts of the contracts such as payment remittance may be accessed exclusively by the owner or the authorized member of the PDL. Moreover, all the entry points to the contract can be examined to prevent unauthorised access. Hackers usually exploit such loopholes or openings to gain access to the contracts.

- Termination clauses:

  Smart Contracts by-definition cannot be destructed but become inactive. Termination clauses allow the contract to stop its execution and become inactive; this prevents the ledger from having eternal contracts. Moreover, after a specific time, a contract may be self-destructible to avoid outdated versions of the contracts and allow the modified new versions.

It is be noted here that Smart Contracts vary in certain ways from legal contracts which cannot self-destruct but may include clauses after which those contracts become ineffective.

5.3.6 Validation

The Smart Contract may be the exact and true representation of the natural language contract and perform only the tasks specified there. In other words, semantic gaps between the expected and the actual execution are important to be eliminated to avoid the wrongdoings of a contract and implement an error-free code. The semantic gaps can be checked at Level 3 of the testing process (Figure 5-2).

5.3.7 User experience testing

A group of users can test a Smart Contract on a sandbox. Their feedback will help in two ways:

1) the future users of the product can comment on the quality of the contracts and future development; and

2) identify the errors and semantic gap in the contracts.

5.3.8 Consumer protection

It is recommended to exercise the disclosure of minimum terms and conditions to transfer liability from the developers to the user. The user may take full responsibility for the protection of sensitive data such as keys as leakage of information can put other PDL members' data at risk.

5.4 Deployment and execution phase

5.4.1 Deployment

Smart Contracts by-design once deployed cannot be changed or amended. Hence, extensive emphasis on careful planning and design has been placed on the earlier stages. In the deployment stage, the contract is installed on a PDL, and it particularly involves the stakeholders such as a mobile operator and a tractor vendor, who agreed on a contract for network services. This stage may not necessarily involve the developers as the deployment can be straightforward if the earlier steps are carried out correctly, and the pre-tested template of a required contract is available.

5.4.2 Execution

Deployed contracts can be executed unlimited times (depends upon the under-lying PDL type) during the execution phase. The execution of a Smart Contract can be parameterized, and non-parameterized depends on the design model and can be performed by any authorized party through an API. Rest APIs can be used here, and the payload can be implementation-dependent.

5.4.3 Termination

Smart Contracts are recommended to be terminated exclusively, or they may be self-destructible after certain a time as may contain critical conditions such as pay-outs. In this case, if a dormant contract exists in a ledger can be exploited by the adversary. The termination of the contract can be done by the contract itself (i.e. destroys itself) or through an API handled exclusively by the stakeholders through the digital-signature mechanism, to ensure security. The termination may exclusively be performed by the owner of the contract, and it is possible that instantiation of one contract terminates the older one.

6 Architectural requirements for Smart Contracts

6.1 Introduction

Smart Contract depends on the PDL type, and their architecture is also dependent on the PDL support. The careful design of the internal architecture of Smart Contracts is important to design a safe and scalable Smart Contract. In this clause, the architectural requirements for a viable Smart Contract are discussed.

6.2 Architectural requirements

6.2.1 Reusability

Since a Smart Contract is a software that can live forever in a PDL, its architecture may be able to provide flexibility for reusability; that is to say, a contract may be generalized enough to be used multiple times. In a Smart Contract, key parameter such as start date, end date, and beneficiary information can be specified to allocate a Smart Contract to several users.

The reusability can prevent the dormant contracts and the PDL being populated, thus helps in scalability.

6.2.2 Self-destruction

As discussed in clause 5, Smart Contract may be destroyed or terminated after some time to avoid dormant or eternal contracts. However, some contracts are not suitable to be destroyed or terminated completely. For example, contracts with some monetary value cannot be terminated because their destruction will cause the customers to lose funds. However, if a contract is some kind of agreement, for example, an agreement between a user and their network service provider, it can include the self-destructive clause.

Self-destruction may have two substates:

1) End of use

2) Management removal, or achieving, if the self-destruct clause allows this. This may run:

   a) On time-out.

   b) On periodic heartbeat.

   c) On explicit management action.

6.2.3 Data ownership

A Smart Contract may comply with GDPR requirements and keep public data only. If a Smart Contract wants to access or keep the private data (i.e. under certain licensing restrictions), the governance of the PDL may take and record appropriate permissions from the owner of the data.

6.3 Reference architecture

6.3.1 Introduction

Smart Contract development may go through three different processes:

1) logic - in which the original purpose of a Smart Contract is defined;

2) algorithm - the code logic and the interpretation of logic to execution; and

3) code - the final code which is a true representation of the initially planned logic.

In Figure 6-1, these processes are illustrated.

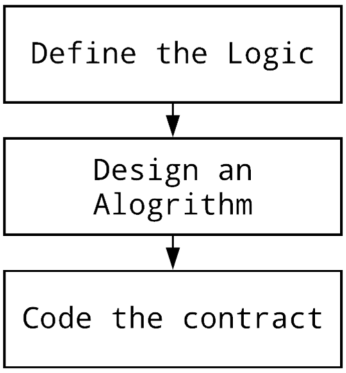

Figure 6-1: Processes of a Smart Contract

6.3.2 Data retrieval in Smart Contracts

A Smart Contract may retrieve data from external sources such as oracles through an API. This access can be in compliance with the governance of the PDL and the country laws such as GDPR.

As discussed in clause 6.2.3, the data added to this Smart Contract may comply with GDPR - the Smart Contract can keep only public data - private data, if added, is informative and comply with regionally agreed and national regulations.

6.3.3 Transactions and transaction dependencies

A Smart Contract deployment and execution creates a transaction which is recorded in the PDL.

It is possible that a Smart Contract transaction is initiated by other Smart Contracts; that is to say that a Smart Contract execution is dependent upon certain prerequisite state of another Smart Contract and triggered by them. Here is some consideration to be taken care of, such as the latency of prerequisite Smart Contracts may delay the execution of future contracts.

The transaction ordering for a Smart Contract is important to be defined in the consensus of the corresponding PDL. It is recommended to adopt specific ordering of transaction inside the base contract (i.e. the contract which will initiate the chain of contracts) to avoid transactions being rejected and cause clutter in the ledger because even the rejected transaction is recorded. Additionally, an appropriate delay may be added to call the next contracts; this approach can mitigate the problem of latency and provide sufficient time for earlier contract transactions to complete their execution.

6.3.4 Smart Contract architecture - Without Smart Contract chaining

Following reference architecture defines a Smart Contract with recommended processes. Organizations may choose their own variations depending on the PDL type and requirements of the use-case and may consider this as an initial design.

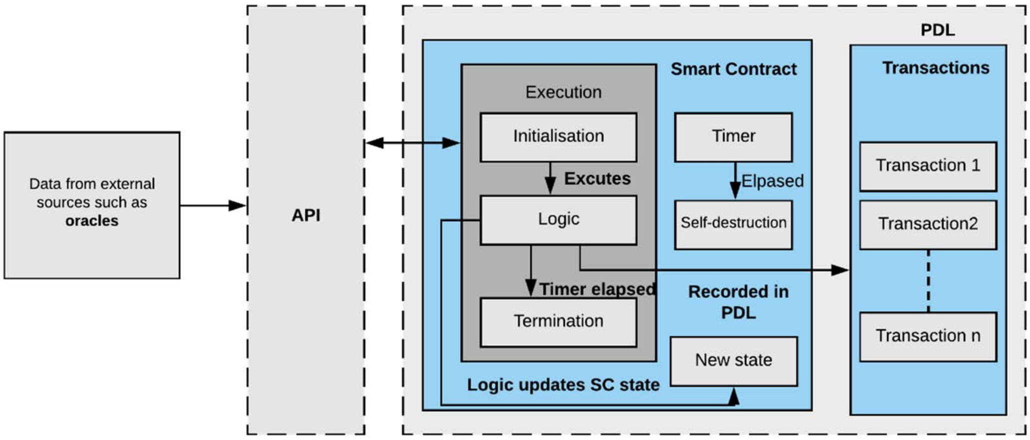

Figure 6-2: Reference Architecture of a Smart Contract without contract chaining

6.3.5 Smart Contract architecture - with contracts chaining

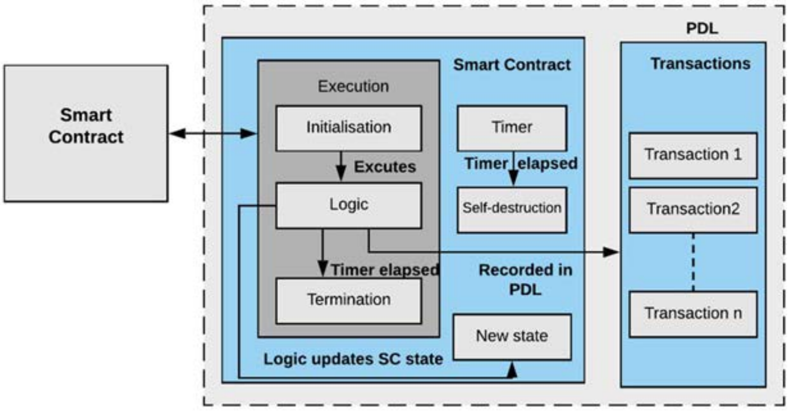

Figure 6-3: Reference Architecture of a Smart Contract with contract chaining

7 Smart Contracts - applications, solutions and needs

7.1 Introduction

Smart Contracts and their properties can be useful in many applications. Smart Contracts can be applied in any DLT scenario where an automated and transparent contractual mechanism is required. However, the limitations and implications of adopting a Smart Contract based solution may be considered. In this clause, some of the possible applications and limitations of Smart Contracts are highlighted along with potential solutions.

7.2 Applications

7.2.1 Introduction

Smart Contracts can potentially be a viable solution for applications where transparency and immutability are a priority. They provide a mechanism to automate the contractual process, track the contract executions, and provide accountability in the contractual process. There are several ways and solutions where Smart Contracts can be applied to achieve the goals mentioned above, and some of them are highlighted here.

7.2.2 ICT Sector

In the ICT sector, there are a number of ways a digital service provider and a customer (business or individual) engage in contracts. For example, Home mobile provider and Visited mobile provider have contracts for roaming services; the services consumed by the customer in the visited location is recorded and sent by the visited provider to the home provider. Smart Contracts can automate this procedure by enabling service providers to create Smart Contracts for such digital services; as soon as the visiting customer consumes the network services of the visited operator, the corresponding Smart Contract is activated and enables instant settlement between the host and the visited provider including the availability of the credit and payments.

Furthermore, mobile operators may not offer the same consistent performance; factors such as congestion in the area and day/time impact the performance [i.3]. This may result in a violation of the SLA between the user and the service provider. In situations where the mobile operators cannot provide the required QoS, possibly due to the congestion, customers may consider getting the services from other operators who offer a service guarantee. These provisions need automaticity and transparency. The customer wants to get the services instantly and automatically. In the scenarios where QoS is of paramount importance (e.g. services for life-relying activities such as remote surgery), strict SLAs are expected be honoured, and if the violation happens, the customer is notified (Transparency) and potentially compensated. Smart Contracts can help to achieve these targets and provide a contractual framework in an untrusted environment. This is achieved through logging of SLA and performance data on a PDL, and applying a Smart Contract to calculate the actual performance against the targets and automatically calculate the penalties according to the SLA where applicable, penalties can be automatically reduced from the invoice on the next billing cycle.

7.2.3 Automated machines/sensors

Automated machinery such as tractors and solar farms are equipped with sensors; these sensors transmit the device data such as engine readings or battery life to the Cloud or command centre, where this information is processed to make future decisions such as capacity planning. Such systems are vulnerable to eavesdropping, replication, and man-in-the-middle attack. The attacker can pretend to be a legitimate device and send erroneous or incorrect data to the command centre, and the valid user can be blamed for sending false/fake information. Such attacks can be mitigated using Smart Contracts, which can be installed on the ledger and while transmitting the sensor data, the unique identifier of the sensor sent along with the data, this information will be recorded as part of Smart Contract execution, which can verify the identity of the sensor. It is expected that data is sent within a quantum-safe encrypted form to mitigate man-in-the-middle attack and eavesdropping.

7.2.4 Automated auctions/sales

Automated auctions are found in almost every field - for example, telecom regulators auction bandwidths to operators. Smart Contracts can help automate this process in such a way that the bandwidth contract is installed on a PDL with predefined parameters. An auction starts and ends with predefined time, and all the bids are recorded in a PDL. This process becomes transparent to all the parties preventing dishonesty both by the bidder and the auctioneers. These bids can be tailored for specific needs for visibility and automated actioning.

7.2.5 Mechanism for access control/certification authority

Smart Contracts may be used as a mechanism for access control; as by definition, they execute automatically, all the access information (e.g. user credentials) can be recorded in a PDL. For example, a Smart Contract can be executed when some access rights are granted by a PDL-based certification authority. This may prevent the future disputes of the data breach and provide a record of all the information exchange and key distribution.

In another example, Certificate Authorities are trusted by the users, and it is possible for malicious parties to act as a CA and issue fake certificates. This can cause users to trust malicious websites and share their personal records and bank information with them. This problem can be mitigated with PDLs by distributing trust between a group of users rather than a single entity and can be compromised only when more than 50 % (or any higher threshold set by the governance) nodes are malicious. As soon as user credentials are allocated, the respective Smart Contract can be executed, and all the relevant information for the certificate is recorded. These credentials may be used to access the controlled data or records (e.g. PDL data). Since the credentials are issued by the group of users in a PDL and their integrity is backed by a transparent mechanism, they can be trusted. Also, it is difficult for malicious users to act as a CA because PDLs are managed by a group of nodes, and all the records (such as public keys) are transparent, so the users can verify the integrity of a website with the PDL.

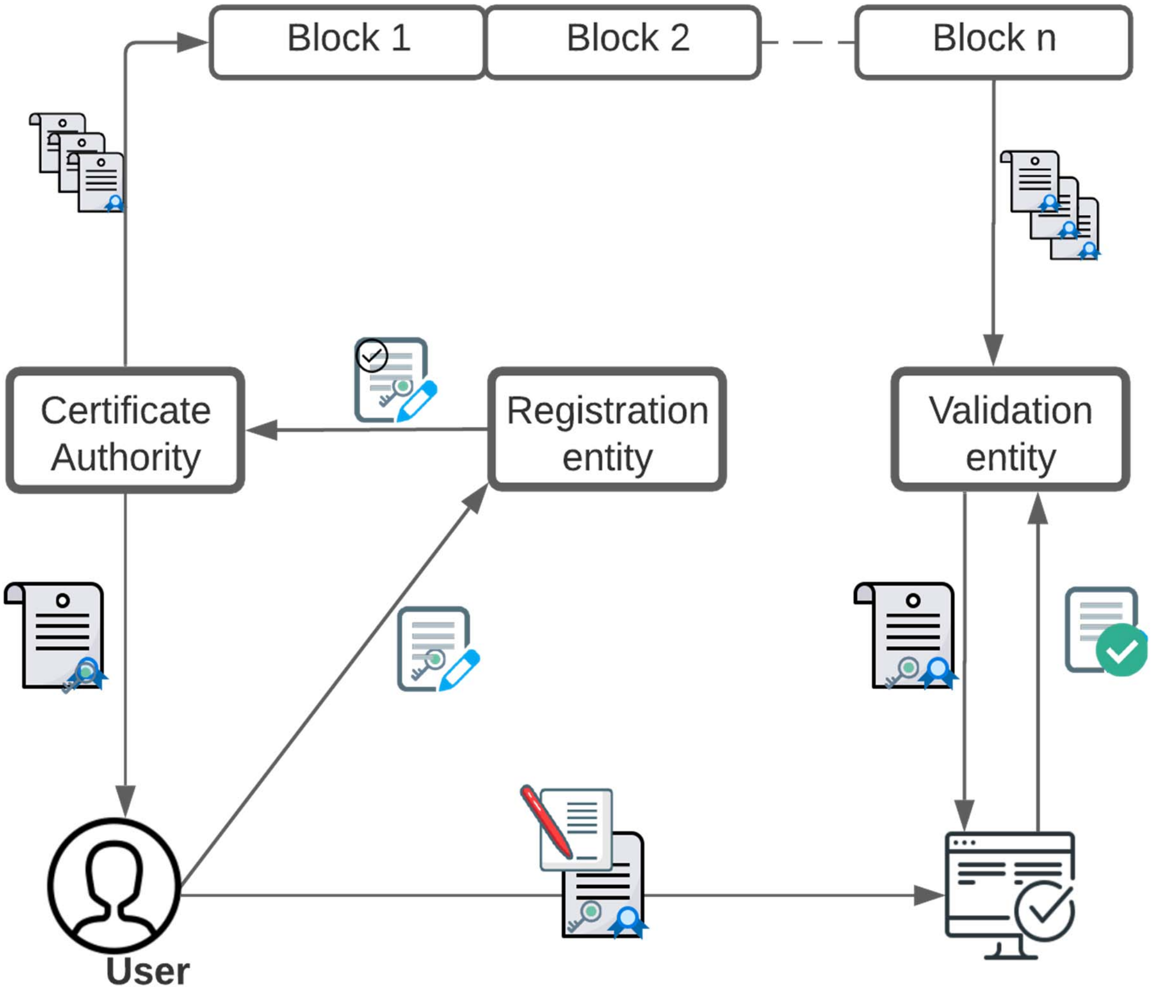

Figure 7-1: Example of PDL based Certificate Authority

Smart Contracts can provide a mechanism for accessing data from a foreign ledger, by distributing authorized keys to the authenticated participants only, in this way, the participants will not need to ask for access keys repeatedly; the key distribution is recorded via Smart Contract to a PDL enabling the records to be updated automatically and transparent to all PDL members. This facilitates the future audit of the access records.

7.3 Solutions

7.3.1 Introduction

Smart Contracts have some limitations such as scalability and immutability, which are already discussed in earlier clauses. In this clause, the possible solutions to these inherent properties of Smart Contracts (e.g. immutability and auto-execution) are discussed. It is to be noted that in certain cases these properties cannot be eliminated but can be mitigated through design and planning.

7.3.2 Scalability

All the executions of a Smart Contract are recorded in a PDL, and removing them from PDL is not possible for a typical PDL. Because Smart Contracts cannot be removed, unused and dormant contracts may live for eternity in the PDL costing PDL node resources. Some potential solutions to manage the scalability problem of PDLs due to Smart Contracts specifically.

7.3.3 Check-point

The Smart Contracts can be installed on side-chains with a check-point to self-destruct after a certain time. A side-chain can record the existence of the contracts in the master-chain before destruction. This can be achieved by introducing a check-point (e.g. a specific date). For example, a side-chain with certain dealings between a telecom operator and a vendor, and once this contract is completed, the chain is destructed, but final settlement transaction may be recorded in the master-chain.

7.3.4 Extensibility

Smart Contracts are immutable; however, they can be extended or revised by adopting the off-chain mechanism. That is to say that the master contract is deployed in a master-chain (or maybe the side-chain acting as master-chain) with the initializing clauses only and include commands which call the logic contract. The logic contracts are separate contracts which may be installed on the same or different (e.g. master-chain or side-chain) PDL or may be installed off-chain (i.e. trusted data structure). Sample architecture for contract chaining is shown in Figure 6-3.

7.4 Security of contracts

Smart Contracts are software and are not web-based; hence the traditional application layer security protocols (such as https) are not applicable to them. Incorrect information can activate Smart Contracts in a manner which may have a negative impact on the ledger and its users. A possible solution is mandating that activation requests for Smart Contracts are always generated from a Trusted Execution Environment (TEE) (Figure 7-2). In Figure 7-2 a Smart Contract based QoS monitoring system is explained where TEE is installed on both the user and the operator, the request to execute a Smart Contract is generated from the user, however, the QoS parameters are reported to the PDL through a TEE which is submitted to the operator through customer's TEE. The detailed procedure is explained in clause 7.5.

7.5 Example: Smart Contracts with QoS monitoring

The architecture explained in Figure 7-2 provides a mechanism of network services allocation using Smart Contracts; the industry can adopt this for an accountable contractual mechanism for network service provisioning. In this architecture, the operators and regulatory authority operate as PDL nodes, and customers (entities who need services) have limited read-only access to the ledger that is, customers, do not take part in the consensus of the ledger. The service contracts along with their Service Level Agreements (SLAs) are recorded (deployed) in the PDL in the form of Smart Contracts.

The goal of this architecture is that the service contracts or Service Level Agreement (SLA) along with QoS metrics provided during the service provision is recorded in a PDL, enabling future audibility and SLA monitoring. To prevent customer or operator being dishonest and reporting wrong QoS parameters, the QoS recording and reporting will be done through TEE only. The annotations in Figure 7-2 are discussed below:

1) All the service contracts from all available operators are advertised on a Distributed Application (DApp) (which works as a marketplace for the service contracts); these service contracts are backed by Smart Contracts stored in the PDL. Like a typical marketplace, a customer can input their requirements and choose a suitable network service offer. The customers may be asked to forward the agreed-upon payment to the chosen operator using traditional means to prevent DDoS attacks on the PDL.

2) Once the customer chooses a service contract, the DApp fills an activation request to the corresponding Smart Contract, transferring the payment due at the same time (which will be payable only if the request is successful). The service request is then encoded as a transaction and sent to the PDL; customer is required to sign this request to prove their approval. As mentioned earlier, the service contracts, as Smart Contracts are already installed on the PDL and ready to accept execution requests.

3) The new transaction containing the activation request is added to the pool of pending requests by the validators (i.e. operators and regulatory authorities), who will eventually accept it, through the distributed consensus algorithm, if well-formed.

4) On successful execution, the respective operator gets notified and can start allocating the resources to provide the requested service.

5) The service from the operator to the customer is being provided. At this stage, the actual QoS is managed by a customer-side and an operator-side Trusted Execution Environment (TEE). The operator-side TEE is called as Performance Monitor.

6) The Performance Monitor records the receipts from the user and send to the PDL inside the corresponding Smart Contract (i.e. their agreement). This allows to verify, and prove, if the SLA has been fulfilled.

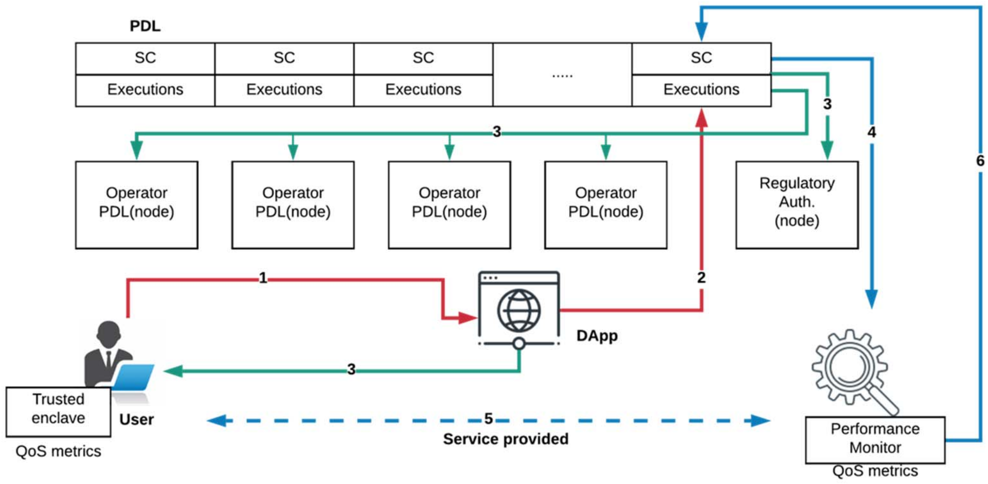

Figure 7-2: Smart Contracts with QoS monitoring

7.6 Needs - Requirements to build a viable system with Smart Contracts

7.6.1 Regulatory aspects

The PDLs' governance may manage the Smart Contracts, the group organizing a PDL can reach a consensus on the regulation of the terms and penalties in case of violation. For example, roaming is currently a challenge for mobile network operators. In current systems, billing in roaming may be a long process and involves several steps such as sending the usage to the home operator to make claims. To resolve this, customers' payment can be directed to the visiting operators through the PDL and invocation of a Smart Contract. This system is only viable when both the participants honour the Smart Contract and in the situations of dispute resolve them as per the governance of the PDL.

7.6.2 Security of the contracts

In Blockchains such as Ethereum, Smart Contracts are publicly available; as per Ethereum consensus a copy of every contract is stored at every node; this may not be a scalable strategy for many real-world applications, where all the participants, even from the same PDL are not involved in every agreement or contract.

In such a situation, a more exclusive mechanism can be adopted, where only the involved participants, may have access to Smart Contracts. To ensure privacy in Smart Contracts, different access rights can be assigned to every participant of the contract. Here, the participants can be direct trading parties or the other stakeholders such as the mediators (in PDL access control mechanisms may prevent security breaches). Another advantage of this strategy is it enables scalability for the nodes.

7.6.3 Secure data feed (oracles)

Smart Contracts usually get data from external sources such as oracle services; sometimes, this data-feed is used by them to start executing specific functions such as payments and penalties. For example, in the telco-sector, the QoS records are submitted to a contract to perform payment functions for the network services provided. It is likely that the participants, such as clients, can tamper with the actual data to benefit themselves. For example, they report wrong QoS metrics to blame the provider for not offering the contractual service. This problem can be tackled at the implementation stage; however, security mechanisms such as the installation of trusted hardware at the customer end, for example Trusted Code Base/Trusted Execution Environments (TEEs) can be adopted after checking implications.

7.6.4 Enforceability

Smart Contracts are self-executable, which means they can automatically execute with the fulfilment of a certain pre-coded condition; When two or more parties internally or externally agree on a contract, they are expected to honour the agreement without any disputes, and if there is any, the stakeholders can come together to resolve the issue as per organization policies.

NOTE: Smart Contracts are enforceable across the borders (i.e. internationally) and can follow the PDL governance policies and the participants' laws. This will be normatively addressed in a future specification regarding the work on the UN regulations for international trade [i.5].

7.6.5 Availability

As Smart Contracts are aiming to be adopted as a contract mechanism for industry, an important consideration for them is to be always available for execution which depends on the transaction speed of the native PDL. If a PDL supports higher transaction speed it also allows more connections to Smart Contracts; the number of requests at the PDL (i.e. transactions) impacts the availability of the PDL hence Smart Contracts.

Hence, to avoid unwanted traffic at the PDL, admission control mechanisms may be applied to ensure legitimate and necessary nodes access the PDL only. For example, a PDL governance may enforce a rule to allow a node to send a certain number of transactions in a specified time only after they are not allowed to send transactions for some specified time, or they can go to hibernation state that is their "Idle-time". The number of allowed transactions and idle time of nodes depends on the use-case and the governance of the PDL, for example, an organization using a PDL-type which allows hundreds of transactions per second may allow more frequent transactions from their users compared to other PDL-type which support tens of transactions per second which can accommodate a smaller number of participants.

7.6.6 Attacks

7.6.6.1 Re-entrancy

Re-entrancy attack happens when the attacker takes hold of the contract and attempts to change the ledger through this contract; one consequence is that they are possibly able to transfer funds to themselves. The most famous example of this attack the DAO attack in 2016, in which the attacker was able to steal 3,6 million Ethers through a re-entrancy attack.

Re-entrancy can be:

1) single function; and

2) cross-function.

In Single Function re-entrancy, the attacker can control only one function and recursively calls the same function to create damage; for example, drain all funds managed by the contract. In Cross-function re-entrancy, the attacker can control functions which share states with other functions. For example, a pay-out contract shares its state with a vulnerable function.

7.6.6.2 Free option problem

This type of problem is well discussed in Plasma blockchain. When two parties, X and Y agree to do some purchase and decide to pay through a Smart Contract, X sends its signed transaction; in the mean-time Y changes its mind and backs-off. In this situation, X has already sent Y the payment for the item, but Y has refused to send the product; in this case, Y has the Free-Option he can take the money without giving the product. In PDLs, this type of attack can be mitigated by the governance and the penalties enforced by them.

7.6.6.3 Denial of capacity attack

Like other distributed systems, PDLs are vulnerable to attacks from malicious parties which can cause Denial of Service (DoS) to legitimate users. For example, since the PDLs allow a finite number of transactions per second, the malicious users can send continuous and redundant service requests from malicious users to the PDL, which can overwhelm the PDL. A global lock of a certain time(possibly a few seconds) can be applied to prevent such happenings. Also, penalties through governance may also prevent such wrongdoings.

8 Threats and limitations of Smart Contracts

8.1 Introduction

This clause discusses two major limitations of Smart Contracts 1) Inter and Intra system threats - These threats are due to an internal and external system of the Smart Contracts and 2) Limitations of a Smart Contract - due to its inherent properties.

8.2 Inter and intra system threats

8.2.1 Introduction

ITU, in its report on DLT [i.3] identified these potential risks to Smart Contract technology:

1) a reliance on a computer system itself that executes the contract;

2) flaws in the Smart Contract code (clause 8.2.5); and

3) the reliance on an external 'off-chain' event or person - to integrate with and execute - the embedded terms of the contract.

Some prominent points and their possible mitigation techniques are discussed below.

8.2.2 Absence of termination clause/self-destruction

In every Smart Contract, a termination function is a fragile entity. If it does not exist or is not programmed with the utmost care, can be active for an indefinite period, which can prove very dangerous. For example, if a contract is meant to be writing vehicle service records to the ledger such as location etc. and this car is sold by the company to another company, the absence of or flaw in termination function can result in this vehicle to continue sending the critical data to the ledger. This is dangerous to the new owner of the car because his information, perhaps critical, is being seen by a third-party; also, for the old owner as this vehicle is still utilizing the ledger and occupying the costly storage. For example, if a contract stipulates payment for a certain period of time and the contract does not expire after that period, the amount will be paid indefinitely. Indeed, the payments can be cancelled by other means such as informing banks to stop the payment, but that is also dependent on the design of the contract. Moreover, if such errors go unnoticed, can potentially result in more significant losses such as the execution of certain terms which may harm the company's reputation.

8.2.3 Admission control

Smart Contracts may be allowed by authorized participants only through stringent access control mechanisms; strong governance can potentially handle this, and consensus agreed by the PDL members. If Smart Contracts' access is not carefully managed, they can become open to malicious users. However, this risk in a PDL is minimum since the participants are usually known and allowed with consensus, yet the risk of a replay attack exists. In such attacks, the malicious party intercepts the communication, and sends a modified data; if an attacker can alter the data such as payment amount or the payee, the payments will be issued by the contract. Admission control mechanisms can ensure that the transactions received by the legitimate client only.

8.2.4 Off-chain and side-chain contracts handling

Smart contracts may be installed off-chain or on side-chains; they can be called external contracts. These external contracts may not have full access the ledger and may be allowed to record some limited information to the main-chain to synchronize with the system only and not allowed to perform specific actions such as access or read other contracts of the main-chains or other side-chains.

However, these external contracts still may have certain write access to the main-chain, as they may be allowed to report their contract status to the main-chain to synchronize with the network. The critical consideration here is that if these side-chains or external data sources (i.e. the data structures maintaining off-chain contracts) are compromised, they can send malicious and erroneous data to the main-chain. Such acts can cause more massive disruptions to the PDL system, such as unauthorised initialization of other contracts and sending false information to the main-chain.

A method to mitigate such problems can be intrusion detection mechanisms installed on all the external sources and the strong accountability imposed by the governance to the management of these data storages (i.e. side-chains and off-chain storages).

8.2.5 Poor exception handling

If syntax and logic errors in a Smart Contract are not thoroughly checked and handled, it can cause an infinite loop or hanged contract; this danger can be mitigated by careful design and testing of contracts, as discussed in clause 5.

8.2.6 Transparency of a PDL

Though private, PDL is still shared among members means that transactions are visible to the members. This can be dangerous when competitors are sharing a ledger, for example in the situation of bidding, the price of bid is recorded as a transaction in the ledger, the competing members can see this value in the ledger and can exploit this vulnerability. This situation can be mitigated by governance such as using a hash instead of actual value or enter only encrypted values in the ledger.

8.2.7 External libraries

Computer software such as Smart Contracts rely on built-in programming language libraries; these third-party libraries are prone to error, and using them may be risky. Furthermore, the malicious party can develop such a library to penetrate in Smart Contracts. Developers may consider security vulnerabilities while using third-party libraries to avoid any dangers to the Smart Contract.

8.3 Limitations

8.3.1 Introduction

Smart Contracts' inherent properties also cause some limitations. This clause outlines these limitations and considerations that need management before the deployment of a viable contract. These limitations are specific to the Smart Contract and dependent on the underlying PDL-type. For example, if some PDL-type with high transaction speed, more Smart Contracts will be executed per second than the PDL-type, allowing fewer transactions per second.

8.3.2 Occupancy

Smart Contracts are software codes, and they are installed on a PDL, which by-definition is immutable. Hence if a Smart Contract is installed on a PDL, it cannot be deleted or amended. As discussed in earlier clauses, there exist mechanisms that allow the contracts to be updated. With such techniques, a new copy of a Smart Contract is installed, then the pointer to the old contract is updated. These techniques do not remove the old contract, and it lives in the ledger but dormant.

If dormant and inactive contracts populate a PDL, it can cause scalability problems over time.

8.3.3 Latency

The key consideration for deploying a Smart Contract is the delay or latency. The latency of a Smart Contract is the time it takes for a contract to get deployed and executed and can be categorized in:

1) deployment latency; and

2) execution latency.

Smart Contracts get compiled on the local machines which can potentially be personal computers; then the request to deploy them is issued by the deployment entity through a transaction. In this situation, the Smart Contract latency is dependent on the compilation of the code and the network delay for a contract request to reach the chain.

Mostly, Smart Contracts get executed more often than deployment. The pre-deployed Smart Contract can be executed by any entity with the right permissions. To execute or invoke a Smart Contract, a transaction is issued by the invoking entity, and this depends upon the factors such as network connection and the congestion at the chain. Moreover, the nodes of the ledger by-design are distributed across the World and computation, and speed limitations of every node add an overhead to the latency in the verification of contract transaction.

The method of deployment and execution discussed here is a high-level picture of the Smart Contract system and is strongly dependent on the underlying chain.

8.3.4 Underlying and Relying ledgers in permissioned context

One of the most important considerations for the industry to adopt Smart Contract technology is that of the underlying ledger. Smart Contracts are deployed on the ledger such as Corda, Ethereum or Hyperledger Fabric. Every ledger is unique in its properties and has different resource requirements. As of the time of writing the present document, there is no system for ledgers to interact with different ledger exist, all the organizations or nodes use the same underlying ledger technology in order to implement the Smart Contract as their contractual mechanism. This is not always possible for several reasons such as economically and feasibly to use same ledger technology: hence, be part of the consortium.

8.3.5 Not every term can be translated to a Smart Contract

Smart Contracts are nonetheless a computer program, and computer programs have very strict rules, such as if this then that or do this until this condition becomes true or false. Nevertheless, in real-world contracts, the conditions are not always this rigid and there is flexibility allowed intentionally by both parties, for example, if a business relationship between two organizations is old and they do want to give each other some discount but not to record in the contract, then it may be difficult to have a Smart Contract. For a Smart Contract, either it is, or it is not, there is no opportunity for a middle ground. However, it is important for parties to be transparent in the contractual process and such bilateral promises which cannot be translated to the code, can be recorded in additional contract field in a plain text or in hash format, this will enable transparency between the participants. Adding this field in a hash form, can be verified later.

8.3.6 Legal uncertainty

PDL is comprised of distributed nodes, which can potentially be spread across the globe. The enforceability of Smart Contracts in different countries can be an issue.

Legal aspects of contracts are beyond the scope of the present document, but geographic regulations and laws such as GDPR still applies and depends on the governance and consortia.

For example, if two parties exist in the same country, the country laws will apply, but in a multi-jurisdictional transaction, it is recommended to follow the UNCITRAL arbitration rules and considerations [i.5].

8.3.7 Intellectual property rights

If a Smart Contract is deployed on a consortia ledger, it is important that parties be aware of the potential exposure of Smart Contract code to other parties, as depending on the ledger, either the source code of the contract, or the compiled version of the contract is shared across the whole distributed ledger. This requires the parties to manage IPR related to the Smart Contract, and potentially include licensing across consortia members or non-disclosure claims in the consortia agreement to meet the required IPR management standards across the different organizations.

8.3.8 Accountability in smart contracts

Following some pre-auditing mechanism to guarantee the completeness of the Smart Contracts, there would be two dimensions:

a) Smart Contracts that are minimal functionalities or security functionality components with the building blocks consensus.

b) Smart Contracts that are for business layers and for development and enhancement proposals.

In terms of functional components, the accountability has cleared by the governance model which may include a mechanism of testing, discoverability issues and mitigation of bugs before the genesis of the PDL which normally occur on testnet period before the mainnet, but it has to be audited before the genesis block of the network which it would be governed.

For the business layers, there is a variety of approach which in permissioned environment, either public or private, have a modular ingredient whereby minimal terms of use are recommended and complete acquaintanceship with the governing body of the PDL however in some cases the accountability could be a private permissioned environment whereby the responsibility and liabilities would be by the perfected interest in business although replicate the usage of the PDL in accordance with the consensus mechanism.

Annex A: Change History

Date	Version	Information about changes
11-2019	0.0.1	Initial draft - added table of contents
12-2019	0.0.2	Added SC introduction, types and some text
01-2019	0.0.3	Cleaned draft
02-2020	0.0.4	Cleaned up after F2F meeting - deleted clause 6
03- 2020	0.0.5	Added lifecycle + clause 9
06-2020	0.0.6	Limitations+cleared after F2F(online) meeting
06-2020	0.0.6	Added architecture
10-2020	0.0.14	Cleaned up version from Edithelp
30-10-2020	0.0.14	Clean up after comments from the group
05-11-2020	0.0.15	Clean up and sent for final draft
15-12-2020	0.0.16	Clean and resolve comments and send for publishing

History

Document history		
V1.1.1	February 2021	Publication