

NANCY

**An Artificial Intelligent Aided Unified Network for Secure Beyond 5G Long Term
Evolution [GA: 101096456]**

Deliverable 5.2

NANCY Security and Privacy Distributed Blockchain-based Mechanisms

Programme: HORIZON-JU-SNS-2022-STREAM-A-01-06

Start Date: 01 January 2023

Duration: 36 Months



**Co-funded by
the European Union**

6G SNS

NANCY project has received funding from the Smart Networks and Services Joint Undertaking (SNS JU) under the European Union's Horizon Europe research and innovation programme under Grant Agreement No 101096456.

Document Control Page

Deliverable Name	NANCY Security and Privacy Distributed Blockchain-based Mechanisms
Deliverable Number	D5.2
Work Package	WP5 ‘Security, Privacy, and Trust Mechanisms’
Associated Task	T5.2 ‘Security & Privacy Blockchain-based Mechanisms’, T5.3 ‘Distributed & Decentralised Blockchain’
Dissemination Level	Public
Due Date	28 February 2025 (M26)
Completion Date	27 February 2025
Submission Date	28 February 2025
Deliverable Lead Partner	NEC
Deliverable Author(s)	Wenting Li, Javier de Vicente, Giorgia A. Marson, Sébastien Andreina (NEC), Grigorios Kalogiannis (DRAXIS), Ilias Theodoropoulos (8BELLS), Stratos Vamvourellis (8BELLS), Rodrigo Asensio (UMU), George Niotis (SID), Konstantinos Kyranou (SID), George Michoulis (SID), George Andronikidis (SID), George Tziolas (SID), Anastasios Lytos (SID)
Version	1.0

Document History

Version	Date	Change History	Author(s)	Organisation
0.1	29 October 2024	Initial version and table of contents	Javier de Vicente	NEC
0.2	20 December 2024	Initial contributions by partners	Wenting Li, Giorgia Marson, Javier de Vicente, Grigorios Kalogiannis, Ramon Sanchez-Iborra, Rodrigo Asensio, George Niotis, Konstantinos Kyranou, George Michoulis Cristina Regueiro, Borja Urquizu	NEC, TEC, UMU, DRAXIS, SID
0.3	22 January 2025	Reviewed and extended contributions	Javier de Vicente,	NEC
0.31	6 February 2025	Contribution regarding the smart pricing component	Ilias Theodoropoulos, Stratos Vamvourellis	8BELLS
0.4	12 February 2025	Final version before external review (see Internal Review History)	Wenting Li, Javier de Vicente, Grigorios Kalogiannis, Ramon	NEC, TEC, UMU, DRAXIS, SID

			Sanchez-Iborra, Rodrigo Asesio, George Niotis, Konstantinos Kyranou, George Michoulis, George Andronikidis, George Tziolas, Anastasios Lytos, Cristina Regueiro, Borja Urquizu	
0.5	24 February 2025	Final version before Quality Manager Revision	Javier de Vicente	NEC
1.0	27 February 2025	Final version after quality revisions	Anna Triantafyllou, Dimitrios Pliatsios	UOWM

Internal Review History

Name	Organisation	Date
Ramon Sanchez-Iborra	UMU	17 February 2025
Konstantinos Kyranou	SID	21 February 2025

Quality Manager Revision

Name	Organisation	Date
Anna Triantafyllou, Dimitrios Pliatsios	UOWM	27 February 2025

Legal Notice

The information in this document is subject to change without notice.

The Members of the NANCY Consortium make no warranty of any kind about this document, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

The Members of the NANCY Consortium shall not be held liable for errors contained herein or direct, indirect, special, incidental, or consequential damages in connection with the furnishing, performance, or use of this material.

Co-funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or SNS JU. Neither the European Union nor the SNS JU can be held responsible for them.

Table of Contents

Table of Contents	4
List of Figures.....	6
List of Tables.....	7
List of Acronyms	8
Executive summary	9
1 Introduction.....	10
1.1 Relation to Other Tasks and Deliverables	10
1.2 Structure of the Document	11
2 The NANCY Blockchain	12
2.1 Introduction to Blockchain Technologies and Hyperledger Fabric	12
2.1.1 Introduction to Blockchain	12
2.1.2 Decentralization and Enhanced Security.....	12
2.1.3 Applications beyond Cryptocurrency	13
2.1.4 Hyperledger Fabric	15
2.1.5 Attack vectors on Hyperledger Fabric	18
2.2 Introduction to the NANCY Blockchain and its Core Components.....	20
2.2.2 NANCY Blockchain Workflows and Relation to the NANCY Architecture	23
2.3 The NANCY Blockchain Core Components	33
2.3.1 Smart Contract-based Components.....	33
2.3.2 Oracles and Non-Smart Contract-based Components	40
2.4 Other Features of the NANCY Blockchain	45
2.4.1 Blockchain Monitoring Dashboard.....	45
2.4.2 Blockchain Scalability Mechanisms	46
2.4.3 Data Integrity Mechanisms	54
2.4.4 Tokenization of Digital Assets.....	54
2.4.5 Monitoring and Verification of the Transactions	59
3 NANCY ID Management Tools.....	62
3.1 Introduction to SSI and wallets	62
3.1.1 DID Registry and VC (Revocation) Registry	63
3.2 The NANCY Wallet.....	65
3.2.1 PQC Signature Capabilities of the NANCY Wallet.....	65
3.2.2 SSI Capabilities of the NANCY Wallet	66
3.2.3 Architecture and Interfaces.....	66
3.3 Further Mechanisms for Ensuring the Security and Privacy of the Users.....	96



- 4 Research on Additional Security Mechanisms 101
 - 4.1 Protection of Smart Contracts..... 101
 - 4.2 Lightweight Clients 104
 - 4.3 Impact of Decentralization on Security 106
- 5 Conclusions and Future Work 110
 - 5.1 Conclusions..... 110
 - 5.2 Future work 110
- References..... 112

List of Figures

Figure 1. Energy consumption vs. difficulty	13
Figure 2: Fabric transaction flow example	18
Figure 3. The updated NANCY architecture	24
Figure 4. Detail of the inter-operator domain.....	24
Figure 5. Listing services in the marketplace	26
Figure 6. Service selection.....	27
Figure 7. Listing services in the marketplace	29
Figure 8. Service selection inside the offloading and caching workflow	30
Figure 9. Marketplace main interactions	35
Figure 10. Marketplace search status	37
Figure 11. The high-level architecture of DAC component.....	41
Figure 12. Blockchain oracles high-level architecture.....	44
Figure 13 Fabric blockchain explorer	45
Figure 14 Control panel of chain-splitting.....	46
Figure 15 Lifecycle transactions to update channel information.....	46
Figure 16 Transaction throughput and latency	49
Figure 17 Peak throughput achieved by different network size.	49
Figure 18 Pseudo-code of random assignment for organizations between parent chain and child chain	53
Figure 19 Latency of the chain split procedure versus number of organizations in the network.	54
Figure 20. Admin prerequisites	55
Figure 21. Smart contract pseudo-lang	56
Figure 22. High-level architecture of the NANCY Transaction evaluation	61
Figure 23. SSI Architecture with NANCY wallet and NANCY blockchain	62
Figure 24: Architecture of the PQC Signature Solution.....	67
Figure 25: Architecture of the PQC Signature Token	67
Figure 26: NANCY PQC Java SDK	68
Figure 27: Generating a key pair	69
Figure 28: Signature Creation.....	70
Figure 29 Architecture of the interaction between the wallet and the NANCY blockchain	71
Figure 30: ABC entities and interaction.....	97
Figure 31: NANCY ID Key hierarchy and Pseudonym generation.....	98
Figure 32. NANCY p-ABC ID management workflow.....	99
Figure 33 Prevalence of Anchor contracts in all the deployed contracts Solana smart contracts.....	103
Figure 34. Novel light client protocol: performance evaluation.	106
Figure 35: Impact of the number of nodes on the network delay, for different blockchain designs.	108
Figure 36: Impact of the number of nodes on the probability of preserving security, in different blockchain designs.....	109

List of Tables

Table 1: Extended functional requirements.....	20
Table 2. Interface description of smart contract SLA Registry	33
Table 3. Interface description of smart contract Marketplace	37
Table 4. Fabric orderer configuration in configtx.yaml	48
Table 5. Invoke vs Query Functions.....	57
Table 6. Interface description of smart contract DIDRegistry.....	63
Table 7. Interface description of smart contract VCRegistry	64
Table 8. Vulnerabilities and non-security-related issues (distractions) of the review task	102
Table 9. Number of participants found distractions and security vulnerabilities.....	102
Table 10. Code review outcomes	103

List of Acronyms

Acronym	Explanation
API	Application Programming Interface
BFT	Byzantine Fault Tolerance
BS	Base Station
BSI	Bundesamt für Sicherheit in der Informationstechnik
DAC	Digital Agreement Creator
ETSI	European Telecommunications Standards Institute
FIPS	Federal Information Processing Standards
GUI	Graphical User Interface
MKT	Marketplace
NIST	National Institute of Standards and Technology
PFX	Personal Information Exchange
PKCS	Public Key Cryptographic Standards
PoW	Proof of Work
PQC	Post-Quantum Cryptography
QMS	Quantum Management System
SP	Smart Pricing
UE	User Equipment

Executive summary

Blockchain technology, though initially designed for digital currency, is a foundational innovation with the potential to transform data management, security, and transparency across industries. Its decentralized architecture, coupled with cryptographic security and immutability, supports applications that require trust and accountability, such as supply chain tracking, healthcare, and identity management. Permissioned blockchains further expand blockchain’s capabilities by offering enhanced security measures suited for regulated environments, enabling secure and efficient data sharing. As blockchain technology continues to evolve, its implications extend far beyond finance, positioning it as a key driver of innovation in the digital economy and beyond.

By enhancing security, decentralizing resource management, and promoting transparency, blockchain provides a robust foundation for the future of B5G services. It ensures a secure, scalable, and user-centric environment that meets the complex demands of next-generation telecommunications – such as the case of the scenarios addressed in NANCY.

Functionally, the NANCY Blockchain and its core components realise the inter-operator domain of the NANCY system (please refer to D6.1 ‘B-RAN and 5G End-to-end Facilities Setup’). Specifically, they enable service handovers among operators by means of secure exchange and signature of SLAs. This document describes these components, their interfaces and workflows, and their relation within the NANCY Architecture. Data integrity and scalability are necessary features in NANCY and are also reported.

The NANCY system also offers significant *privacy* advantages for the users through Decentralized Identifiers (DID) and other approaches. Specifically, Self-Sovereign Identity (SSI) mechanisms are used, described in this deliverable. They enable any user to seamlessly generate a digital identity that is decoupled from its own formal public identity, and for which it has total control during the lifecycle of said digital identity (generation, binding of any data as attributes, deletion, etc.). NANCY is also proof-driven and uses decentralized management mechanisms on top of the NANCY Blockchain for managing the associated identity services (e.g., registration, management and control of associated cryptographic verification data, etc.).

1 Introduction

Blockchain and self-sovereign identity (SSI) mechanisms are increasingly recognized as foundational technologies to ensure privacy and security in B5G service provisioning. As B5G networks aim to interconnect vast numbers of devices, users, and services with unprecedented speed and low latency, they face high risks of data breaches, cyberattacks, and privacy violations. Blockchain, a decentralized and tamper-proof ledger, can provide a transparent, immutable record of transactions across the network. By distributing data across numerous nodes and securing it cryptographically, blockchain enhances trust, minimizes single points of failure, and enables traceability without compromising user privacy.

Blockchain-based marketplaces are decentralized digital platforms where buyers and sellers can exchange goods, services, or digital assets directly without the need for intermediaries like banks, e-commerce platforms, or brokers. These marketplaces are used in totally different sectors. For instance, NFT Marketplaces like OpenSea and Rarible, are used by digital artists and collectors to exchange unique digital assets. On the other hand – and much more importantly for NANCY – Supply Chain Marketplaces are platforms that enable traceability and fair-trade verification, allowing consumers to verify the origin of the products they buy. By leveraging blockchain technology, these marketplaces enhance transparency, security, and efficiency through smart contracts, peer-to-peer transactions, and distributed ledger technology. In NANCY, we use a blockchain-based marketplace to securely register, select and buy B5G services from verified providers, and to transparently agree on Quality of Service (QoS) service levels that are enforced in the system through blockchain events and signed SLAs. These SLAs are recorded immutably in the NANCY permissioned ledger, which can be audited by other components in the architecture. This transparency fosters trust among remote participants.

SSI, a user-centric approach to digital identity management, allows individuals to own, control, and selectively share their credentials without relying on centralized authorities. This is crucial in a B5G environment, where personal data will be generated at large scales. SSI ensures that users can verify their identities in a privacy-preserving manner, reducing the need for data collection by service providers and limiting the exposure of personal information to unauthorized entities. Through blockchain-based SSI, individuals can manage access to their data across multiple services securely, protecting against identity theft and misuse.

In NANCY, Blockchain and SSI align with the privacy-by-design principles fundamental to B5G. By enabling secure, decentralized authentication and authorization, they lay the groundwork for trust and compliance with privacy regulations. These mechanisms are essential for safeguarding user data in a highly interconnected, data-rich B5G ecosystem, ultimately enhancing user autonomy and security in a digital world where privacy remains a growing concern.

1.1 Relation to Other Tasks and Deliverables

This deliverable is associated with tasks T5.2 ‘Security and privacy blockchain-based mechanism’ and T5.3 ‘Decentralised blockchain’ inside WP5. Specifically, in T5.2 (1) PQC mechanisms are used for signing on the blockchain, (2) ID management tools are developed (in particular SSI), and (3) other Blockchain-based security solutions are investigated (e.g. data integrity, scalability and protection of smart contracts). In T5.3 (1) the NANCY blockchain is developed and deployed (wallets and ledger) together with other blockchain-based key components such as the (2) Marketplace and the (3) Digital Agreement Creator. T5.3 works in close alignment with T5.2. Additionally, T5.1 delivers crucial

outcomes for T5.2, specifically the PQC Signature modules that enable the NANCY Wallet to sign, using a PQ-safe key, a service level agreement (SLA) on the NANCY Blockchain. Additionally, outcomes from WP4 – reported in D4.1 - are used in T5.3 for designing the inter-operator domain workflow.

Outcomes from T5.2 and T5.3 are integrated and used in several WP6 demonstrators (refer to WP6 documentation for details), while its workflows for service selection and agreement incorporate WP4 outcomes (namely the SLA model and task offloading workflow of T4.1 (see D4.1) or the Smart Pricing module of T4.5) and in return can also inform other components of the NANCY system, external to WP5, so they e.g. enforce a given SLA or perform measurements to check its compliance.

1.2 Structure of the Document

Beyond its Executive Summary and Introduction sections, the rest of the document is structured as follows:

- **Section 2 – The NANCY Blockchain** presents an introduction to blockchain technologies and Hyperledger Fabric in particular, followed by descriptions of the NANCY Blockchain, its core components and workflows, and their relation to the NANCY Architecture. Both on-chain and off-chain components are presented, for the sake of completeness, together with their architecture and interfaces. Additionally, other features of the NANCY Blockchain are presented at the end of the section.
- **Section 3 – NANCY ID Management Tools** delves into the Self-Sovereign Identity and other privacy-oriented mechanisms in place at the NANCY Wallet. It also describes the NANCY Wallet itself and its PQC Signature and SSI capabilities.
- **Section 4 – Research on Additional Security Mechanisms** is a summary of important research that has been carried out in the framework of NANCY, covering (1) the protection of Smart Contracts, (2) Lightweight clients and (3) the impact of decentralization on security.
- **Section 5 – Conclusions and Future Work** summarizes major developments and introduces future research beyond NANCY in the fields of T5.2 and T5.3.

2 The NANCY Blockchain

This section presents an introduction to blockchain technologies and Hyperledger Fabric in particular, followed by descriptions of the NANCY Blockchain, its core components and workflows, and their relation to the NANCY Architecture. Additionally, other features of the NANCY Blockchain are presented at the end of the section.

2.1 Introduction to Blockchain Technologies and Hyperledger Fabric

2.1.1 Introduction to Blockchain

Blockchain technology, initially popularized by its association with cryptocurrency, represents an innovative approach to data management, information security, and digital transactions. Blockchain operates as a distributed ledger system where data is stored across multiple nodes, creating a secure, transparent, and tamper-resistant record of information [1] [2]. By design, blockchain technology offers unique characteristics such as decentralization, immutability, and cryptographic security, making it highly versatile and applicable across industries like supply chain management, healthcare, and identity verification.

Blockchain's underlying structure consists of a sequential series of "blocks" containing data batches, each linked to its predecessor through cryptographic hashes. This chain is distributed across a decentralized network of nodes, each holding a complete or partial copy of the ledger [3]. Unlike traditional databases, where data is stored and managed centrally, blockchain data is nearly immutable. Once recorded, data on a blockchain cannot be altered without network-wide consensus, making blockchain highly suitable for applications that require secure and trustworthy record-keeping [4].

2.1.2 Decentralization and Enhanced Security

Decentralization is a cornerstone of blockchain technology, providing security, resilience, and transparency in data management. In a decentralized blockchain network, data is replicated across numerous nodes, eliminating single points of failure and reducing the risks associated with central data storage. Consensus algorithms like Proof of Work (PoW) or Proof of Stake (PoS) enable network-wide agreement on the validity of transactions without relying on centralized control. For instance, PoW, used in Bitcoin, requires substantial computational power, making tampering difficult. In Ethereum and other systems, PoS encourages users to validate transactions based on their stake, ensuring an honest network [5]. These systems rely on probabilistic consensus algorithms which eventually guarantee ledger consistency to a high degree of probability but are still vulnerable to divergent ledgers ("forks").

However, the mentioned consensus algorithms mainly apply in permissionless blockchains (i.e., public blockchains open to all users), which should not be the case in a B5G environment. The enhanced security offered by decentralization also extends to permissioned blockchains, which are particularly valuable for industries requiring strict data governance. Unlike public blockchains, permissioned blockchains restrict access to specific participants, allowing organizations to control and monitor who can contribute to the ledger. These blockchains implement stringent access control, often enforced through cryptographic measures and multi-factor authentication, ensuring that only verified participants can interact with the network. Permissioned blockchains also support fine-grained auditing capabilities, making them suitable for regulated industries like finance, supply chain, and healthcare, where data privacy and compliance with GDPR is paramount [6].

Furthermore, permissioned blockchains use consensus mechanisms optimized for private networks, such as Practical Byzantine Fault Tolerance (PBFT) or Raft, which ensure high transaction speeds and security without excessive computational requirements. In PBFT, for example, consensus is reached as nodes communicate their agreement on each transaction, ensuring data validity even in the presence of some malicious nodes [7]. This mechanism minimizes the risk of tampering, as attackers would need to control a majority of nodes, a challenging and detectable feat in permissioned environments. In this way, permissioned blockchains provide a balance between decentralization and security, offering robust protections against data breaches and unauthorized access while maintaining operational efficiency [3]. Energy wise, one must understand that resource consumption depends heavily on the type of blockchain and its consensus mechanism. Lightweight blockchains can add marginal overhead, but energy-intensive blockchains can drastically increase power consumption. In the case of NANCY, our lightweight-consensus permissioned blockchain reduces energy consumption by more than 99.99% when compared to Bitcoin's proof-of-work (see Figure 1). NANCY uses fastBFT as consensus mechanism, which behaves similarly to PoTS and PRAOS [8] in terms of energy consumption.

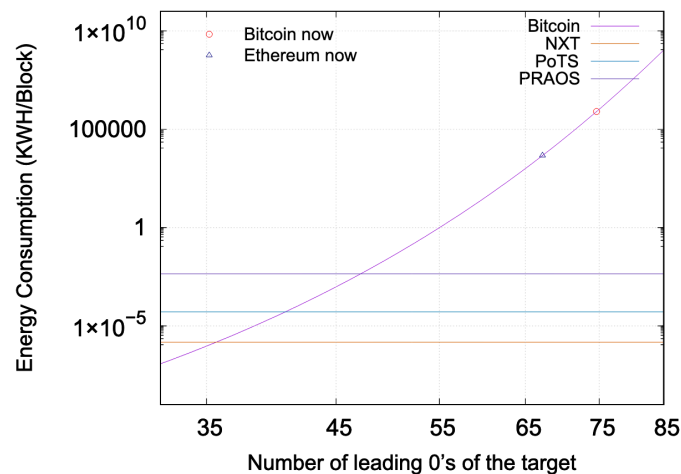


Figure 1. Energy consumption vs. difficulty ¹

Beyond enhanced security, the transparency afforded by blockchain's decentralized structure fosters trust among participants, enabling secure data sharing across complex ecosystems. For instance, in supply chain management, permissioned blockchains facilitate real-time visibility, tracking goods from suppliers to end consumers and reducing fraud and counterfeiting [9]. With verifiable data trails accessible only to authorized parties, permissioned blockchains ensure that shared data remains secure and trustworthy, providing a significant advantage over traditional databases. This is also of importance in a B5G application, where SLAs can be traced and operators and consumers must be trusted.

2.1.3 Applications beyond Cryptocurrency

While blockchain's association with digital currency is well-known, the technology is reshaping numerous sectors by addressing core issues like data integrity, security, and accountability. In healthcare, blockchain offers solutions to fragmented data systems by enabling unified and secure patient records. Traditionally, healthcare information is spread across different systems, resulting in challenges with data interoperability and patient privacy. Blockchain allows for the creation of a secure, distributed record that patients and authorized providers can access across institutions, improving care coordination and reducing administrative delays. The immutability of blockchain also

¹ Source: <https://eprint.iacr.org/2018/1135.pdf>

ensures that medical records remain tamper-proof, a feature critical for protecting patient data and maintaining privacy standards [10].

Supply chain management is another field where blockchain's transparency and traceability have transformative potential. Counterfeiting, fraud, and inefficiencies in global supply chains have long challenged industries from pharmaceuticals to luxury goods. By implementing blockchain, companies can create an immutable, end-to-end record of product movement, enhancing visibility and accountability across supply chain stages. IBM's Food Trust blockchain, for example, enables real-time tracking of food products, which helps reduce food waste, prevent contamination, and ensure freshness upon delivery. Blockchain's capacity for recording and sharing verifiable information across stakeholders has proven particularly valuable for improving quality control and trust among manufacturers, suppliers, and consumers [9].

Blockchain technology also offers significant advancements in identity verification and data privacy. Traditional identity management systems, often controlled by centralized authorities, are vulnerable to fraud and data breaches. In contrast, blockchain enables decentralized identity management, empowering users to control and share their credentials securely without relying on third-party verification. This approach minimizes data exposure, as users can selectively share information with service providers without revealing unnecessary personal details (more on this in Section 3). Blockchain-based identity solutions like uPort and Sovrin offer self-sovereign identities, giving individuals greater control over their digital identities while providing enhanced security and privacy protections [11].

Moreover, blockchain's potential for secure, tamper-resistant voting systems has captured the interest of governments and organizations alike. The transparency and immutability of blockchain make it an attractive option for voting applications, where it can record and verify votes with integrity and without centralized oversight. Estonia has become a leading example, using blockchain-based voting to allow citizens to cast votes online with minimal risk of fraud. Blockchain ensures that each vote is securely stored and independently verifiable, which has improved voter trust and engagement. Beyond national elections, blockchain voting has applications in corporate governance, where stakeholders can participate in decision-making processes with security and transparency [12].

But there are other applications of blockchain, too. In a B5G environment, blockchain can serve as a decentralized backbone, allowing service providers to expose and offer services securely while ensuring users can seamlessly and safely acquire these services. One of blockchain's most valuable contributions to B5G networks lies in its security and resilience. Given B5G's projected massive connectivity and data traffic, blockchain's immutable ledger ensures that transactions are secure, verified, and accessible only by authorized parties. The transparency provided by blockchain allows all participants to verify transactions independently, which is particularly important for B5G's multi-stakeholder environments. For example, smart contracts embedded in blockchain can automate SLAs, ensuring that terms are strictly followed by other parts of the system, reducing dependency on centralized oversight, and increasing trust between service providers and users.

Blockchain also supports decentralized management of network resources, a crucial benefit as B5G networks are expected to handle unprecedented amounts of data and devices. By enabling decentralized transaction handling, blockchain can streamline resource allocation and reduce latency issues. This approach also reduces the need for intermediaries in service transactions, leading to quicker service acquisition and lower operational costs. Blockchain can dynamically allocate resources, adjusting in real-time based on user demand and service requirements, thus supporting the high speed and low latency that B5G applications require [13].

Blockchain’s transparency and traceability also benefit users directly, as they can securely access and pay for B5G services without concerns over data integrity or unauthorized access. By allowing users to verify the service terms and conditions transparently on the blockchain, the technology addresses privacy concerns, which is essential for sensitive applications such as telemedicine, remote monitoring, and financial transactions on B5G networks. Furthermore, and as previously mentioned, blockchain allows users to manage their digital identities securely, giving them greater control over how their data is used and shared across 6G services.

2.1.4 Hyperledger Fabric

The Linux Foundation founded the Hyperledger project in 2015 to advance cross-industry blockchain technologies [14]. Rather than declaring a single blockchain standard, it encourages a collaborative approach to developing blockchain technologies via a community process, with intellectual property rights that encourage open development and the adoption of key standards over time.

Hyperledger Fabric is one of the blockchain projects within Hyperledger. Like other blockchain technologies, it has a ledger, uses smart contracts, and is a system by which participants manage their transactions.

Where Hyperledger Fabric breaks from some other blockchain systems is that it is private and permissioned. Rather than an open permissionless system that allows unknown identities to participate in the network (requiring protocols like “proof of work” to validate transactions and secure the network), the members of a Hyperledger Fabric network enroll through a trusted Membership Service Provider (MSP). Several other permissioned alternatives to Hyperledger Fabric exist, but they are either vendor-dependent (e.g., IBM Blockchain Platform) or do not come with strong enterprise support (e.g., Multichain, Kaleido) or are oriented to particular sectors outside NANCY (e.g., R3 Corda).

Hyperledger Fabric offers several pluggable options. Ledger data can be stored in multiple formats, consensus mechanisms can be swapped in and out, and different MSPs are supported.

Hyperledger Fabric also permits to create channels, allowing a group of participants to create a separate ledger of transactions. This is an especially important option for networks where some participants might be competitors and not want every transaction they make — a special price they’re offering to some participants and not others, for example — known to every participant. If two participants form a channel, then those participants — and no others — have copies of the ledger for that channel.

Hyperledger Fabric is a natural candidate for the NANCY use cases. Here are some of its features [15] and how they can contribute towards realizing a solution for a B5G system:

- **Pluggable Consensus.** Consensus [14] is the process of keeping the ledger transactions synchronized across the network — to ensure that ledgers update only when transactions are approved by the appropriate participants and that when ledgers do update, they update with the same transactions in the same order. Hyperledger Fabric supports a pluggable consensus model, allowing it to be tailored for different performance and fault-tolerance requirements. For B5G use cases, where varying types of devices and applications will interact, this adaptability is a strong advantage.
- **Transaction Performance:** Fabric’s architecture is optimized for high transaction throughput, which is essential for the dense device connectivity expected in B5G, where IoT, edge devices, and mobile units will frequently interact.

- Scalability: As mentioned earlier, Hyperledger Fabric uses “channels” to enable multiple private ledgers within a single network. This segmentation can significantly reduce congestion and improve scalability in a high-density B5G environment, enabling parallel processing across many isolated channels. Also, [15] chaincode execution is partitioned from transaction ordering, limiting the required levels of trust and verification across node types, and optimizing network scalability and performance.
- Integration with PKI: Fabric has robust support for Public Key Infrastructure (PKI), enabling authenticated, secure identities for devices and users.
- Chaincode Functionality: Hyperledger Fabric supports chaincode (its version of smart contracts), which can be implemented in [14] several programming languages. Currently, Go, Node.js, and Java chaincode are supported. Smart contracts provide controlled access to the ledger and support the consistent update of information, but they also enable a whole host of ledger functions (transacting, querying, etc.). In a B5G case, such as NANCY, chaincode could automate tasks like service allocation between devices (see section 2.3.1), SLA creation or authorization of data sharing based on predefined criteria.
- Active Development and Wide Industry Adoption: Fabric's established community² and documentation offer a reliable starting point, plus, being part of the Hyperledger umbrella under the Linux Foundation, Fabric benefits from continuous updates and support.

Before introducing the NANCY Blockchain and its core components, let us delve further into some of Fabric’s key concepts, specifically:

- The shared ledger
- Certificate Authorities
- Peers and Orderers
- The transaction flow

Hyperledger Fabric [14] has a ledger subsystem comprising two components: The world state and the transaction log. Each participant has a copy of the ledger to every Hyperledger Fabric network they belong to. The world state component describes the state of the ledger at a given point in time. It is the database of the ledger. The transaction log component records all transactions which have resulted in the current value of the world state; it is the update history of the world state. The ledger, then, is a combination of the world state database and the transaction log history.

Everything that interacts with a blockchain network, including peers, applications and admins, acquires their organizational identity from their digital certificate and their Membership Service Provider (MSP) definition. Certificate Authorities (CAs) [16] play a key role in the network because they dispense X.509 certificates that can be used to identify components as belonging to an organization. Certificates issued by CAs can also be used to sign transactions to indicate that an organization endorses the transaction result – a precondition of it being accepted onto the ledger.

Different components of the blockchain network use certificates to identify themselves to each other as being from a particular organization. That’s why there is usually more than one CA supporting a blockchain network – different organizations often use different CAs. But also, certificates issued by CAs are at the heart of the transaction generation and validation process. Specifically, X.509 certificates are used in client application transaction proposals and smart contract transaction responses to

² <https://www.lfdecentralizedtrust.org/members>

digitally sign transactions. Subsequently, the network nodes who host copies of the ledger verify that transaction signatures are valid before accepting transactions onto the ledger.

Peers [16] are a fundamental element of the network because they host ledgers and chaincode (which contain smart contracts) and are therefore one of the physical points at which organizations that transact on a channel connect to the channel. A peer can belong to as many channels as an organization deems appropriate. But Hyperledger Fabric [17] features another type of node called an *orderer* (also known as an “ordering node”) that effectively does the transaction ordering, which along with other orderer nodes forms an *ordering service*. Because Fabric’s design relies on deterministic consensus algorithms, any block validated by the peer is guaranteed to be final and correct. Ledgers cannot fork the way they do in many other distributed and permissionless blockchain networks. Orderers also enforce basic access control for channels, restricting who can read and write data to them, and who can configure them.

Now that peers and orderers have been introduced, let us explain how the transaction flow works. Specifically, [17] applications that want to update the ledger are involved in a process with three phases that ensure all the peers in a blockchain network keep their ledgers consistent with each other. In the first phase, a client application sends a transaction proposal to the Fabric Gateway service, via a trusted peer. This peer executes the proposed transaction or forwards it to another peer in its organization for execution.

The gateway also forwards the transaction to peers in the organizations required by the endorsement policy³. These endorsing peers run the transaction and return the transaction response to the gateway service. They do not apply the proposed update to their copy of the ledger at this time. The endorsed transaction proposals will ultimately be ordered into blocks in phase two.

With the successful completion of the first transaction phase (proposal), the client application has received an endorsed transaction proposal response from the Fabric Gateway service for signing. For an endorsed transaction, the gateway service forwards the transaction to the ordering service, which orders it with other endorsed transactions, and packages them all into a block. Ordering service nodes receive transactions from many different application clients (via the gateway) concurrently. These ordering service nodes collectively form the ordering service, which may be shared by multiple channels.

The third phase of the transaction workflow involves the distribution of ordered and packaged blocks from the ordering service to the channel peers for validation and commitment to the ledger.

³ Every smart contract inside a chaincode package has an endorsement policy that specifies how many peers belonging to different channel members need to execute and validate a transaction against a given smart contract in order for the transaction to be considered valid. Hence, the endorsement policies define the organizations (through their peers) who must “endorse” (i.e., approve of) the execution of a proposal. See <https://hyperledger-fabric.readthedocs.io/en/release-2.5/policies/policies.html>

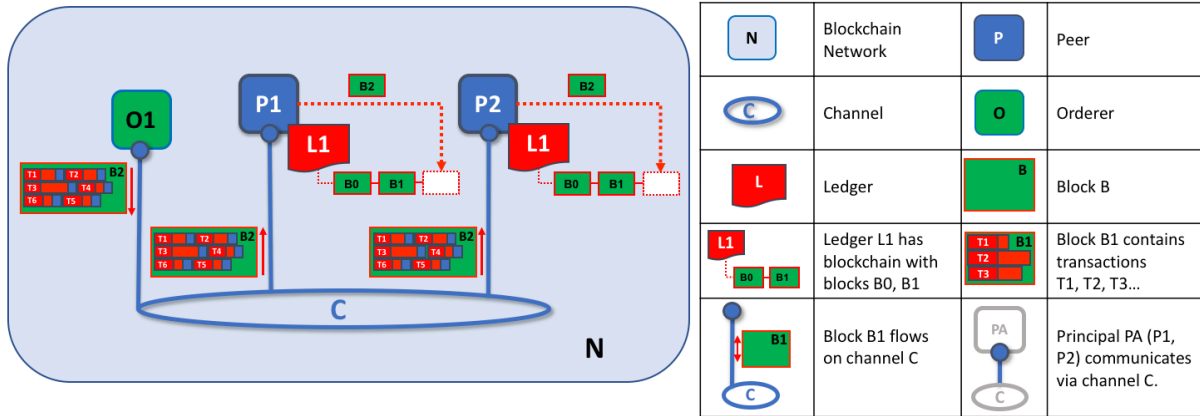


Figure 2: Fabric transaction flow example ⁴

In the example shown in Figure 2 [17], orderer O1 distributes block B2 to peer P1 and peer P2. Peer P1 processes block B2, resulting in a new block being added to ledger L1 on P1. In parallel, peer P2 processes block B2, resulting in a new block being added to ledger L1 on P2. Once this process is complete, the ledger L1 has been consistently updated on peers P1 and P2, and each may inform connected applications that the transaction has been processed.

2.1.5 Attack vectors on Hyperledger Fabric

The use of Hyperledger Fabric drastically reduces the attack vectors found in other permissionless blockchains. However, no blockchain design is completely immune. These are our primary attack vectors and possible mitigations. The reader should keep in mind nevertheless that in most cases we are assuming potential malicious behaviour of nodes that are registered and either paying or receiving funds for services, which is a rather ambitious assumption.

- Vulnerable Smart Contracts (Chaincode): Vulnerable (or even malicious) smart contracts can cause unauthorized data manipulation or access within the network. To mitigate this, some strategies are proposed:
 - Conduct thorough code reviews and security audits of chaincode.
 - Use containerization (e.g., Docker) to isolate chaincode execution.
 - Apply runtime policies, such as limiting resource consumption (CPU, memory) of chaincode.
 - Leverage automated tools for vulnerability scanning in chaincode.
- Identity and Certificate Compromise: Although less probable, attackers could target compromised wallets to steal certificates to impersonate users or nodes. To mitigate this, some strategies are proposed:
 - Use hardware security modules (HSMs) such as TEEs to protect private keys.
 - Enforce multi-factor authentication (MFA) for administrative access.
 - Regularly rotate certificates and implement certificate revocation policies.
 - Monitor for suspicious activity using security information and event management (SIEM) systems.
- Distributed Denial of Service (DDoS) Attacks: Improbable, since the governance and access control of a permissioned blockchain are stronger to those found in a public blockchain, here

⁴ Source: https://hyperledger-fabric.readthedocs.io/en/release-2.5/orderer/ordering_service.html

attackers could flood specific nodes, such as orderer or peer nodes, to overwhelm the network and disrupt its functionality. To mitigate this, some strategies are proposed, which include the use of rate-limiting mechanisms and firewalls to prevent excessive requests or to employ load balancers to distribute traffic evenly across nodes. The NANCY Hyperledger Fabric uses a membership service provider solution so that all users are authenticated and all communication channels are secure (i.e., TLS). Therefore, attacks to public blockchains such as Sybil attack or DoS attack do not practically apply in our case.

- Unauthorized Access: Here, external actors may try to tamper with data stored in the blockchain or the state database. To mitigate this, some strategies are proposed:
 - Use fine-grained access control policies through Fabric's Membership Service Provider (MSP).
 - Encrypt data at rest and in transit using TLS/SSL.
 - Implement channel isolation by ensuring private data is only shared with authorized participants.
 - Regularly audit and validate the integrity of data using hashes stored on the blockchain.
- Side-Channel Attacks. In a side-channel attack, a malicious user targets physical or behavioral characteristics of a hardware system instead of directly attacking cryptographic algorithms. Although this attack is not actually a blockchain-related attack, it can have an effect on the blockchain data. Some strategies for mitigation can include:
 - Use constant-time algorithms to prevent timing-based side-channel leaks.
 - Monitor and limit excessive resource consumption by chaincode or peers.
- Orderer node compromised: Since the orderer service is critical for maintaining consensus, if compromised it can lead to disruptions or manipulation of transaction ordering. Some mitigation strategies:
 - Use Raft or BFT (Byzantine Fault Tolerance) consensus mechanisms to reduce reliance on a single node. In fact, the NANCY consensus mechanism is fastBFT.
 - Apply strict access controls to limit who can interact with the orderer node.
 - Monitor orderer logs for anomalies and implement tamper-evident logging.Attacks such as double-spending or 50% attack only affect certain types of blockchain consensus protocols such as PoW. In our case, we use a scalable and yet provably secure consensus protocol called fastBFT. It tolerates $N/2-1$ faulty nodes in the network. But unlike the 50% attack, in which the attack comes from anonymous nodes, all nodes must first be identified and granted access to our system. Therefore, the assumption of honest majority is more reasonable. Moreover, unlike PoW, fastBFT provides immediate finality and therefore eliminates the problem of double-spending.
- Replay Attacks are based on malicious reuse of valid transactions to disrupt the network or cause inconsistencies. Some mitigation measures (which in fact are already implemented in NANCY) include:
 - Ensure each transaction includes a unique nonce to prevent replay.
 - Validate transactions using digital signatures.

By addressing these potential vulnerabilities proactively, a future implementation of NANCY can significantly enhance the security of its Hyperledger Fabric-based blockchain network.

2.2 Introduction to the NANCY Blockchain and its Core Components

2.2.1.1 Functional Requirements

The NANCY Blockchain and core components developed in tasks T5.2 and T5.3 must fulfil the functional requirements depicted in Table 1 (adapted and extended from NANCY D2.1):

Table 1: Extended functional requirements

Initial Functional Requirement (D2.1)	Extension/Update (D5.2)
Security and privacy of users and devices	<p>The NANCY blockchain should offer significant security advantages for the system through (1) a permissioned blockchain structure, which restricts network access to verified participants only. This model aligns with the high-security needs of B5G, enabling secure data exchanges between diverse devices and stakeholders within the network [18].</p> <p>The NANCY blockchain should offer significant security advantages for the system through (2) a modular architecture supporting robust identity and access management via Public Key Infrastructure (PKI), ensuring that only authorized devices can access or share sensitive data, a feature essential for the integrity of 6G networks [19].</p> <p>The NANCY blockchain should offer significant security advantages for the system through (3) private channels and data collections, allowing sensitive information to be shared selectively and securely among authorized parties, minimizing the risk of data breaches and unauthorized access in a highly interconnected environment and (4) by supporting decentralized, authenticated transactions, reducing reliance on central points of control, which mitigates vulnerabilities to single points of failure. This is a critical consideration for securing complex, distributed B5G systems [20].</p> <p>The NANCY system should offer significant privacy advantages for the users through (5) Decentralized Identifiers (DID) or similar approaches. The approach used should enable any user to seamlessly generate a digital identity that is decoupled from its own formal public identity, and for which it should have total control during the lifecycle of said digital identity (generation, binding of any data as attributes, deletion, etc.). Additionally, the system should be proof-driven and use decentralized management mechanisms for managing the associated identity services (e.g. registration, management and control of associated cryptographic verification data, etc.).</p>
Automated and traceable secure transactions, by means of smart contracts	<p>The NANCY blockchain should offer automated, traceable, and secure transactions by utilizing smart contract functionalities, which automate transaction workflows based on predefined business rules.</p> <p>Each transaction (1) should adhere to specific criteria before execution, reducing manual intervention and improving efficiency within the NANCY applications. Additionally, (2) the NANCY ledger should record each transaction, providing a traceable and verifiable audit trail essential for compliance and security in complex, high-frequency B5G interactions [20].</p>

	<p>By assigning unique digital identities to each participant (see previous row), the NANCY blockchain must also (3) enhance transaction traceability and accountability, linking each action to an authenticated source.</p>
<p>Data integrity and security mechanisms through a Practical Byzantine Fault Tolerance (PBFT) consensus.</p>	<p>The NANCY blockchain should enhance the data integrity of transactions by design. Firstly, (1) it should feature a permissioned network model that restricts participation to verified, trusted entities, reducing the risk of malicious actors tampering with data.</p> <p>Additionally, (2) an endorsement policy mechanism must require that a specified number of trusted peers validate a transaction before it is committed to the ledger, ensuring that only verified, consensus-backed data entries are recorded. PBFT algorithms must be considered for consensus in the NANCY blockchain.</p> <p>Thirdly, (3) the ledger itself must be immutable and cryptographically secured (e.g. each block must contain the certificate and signature of the block creator which is used to verify the block by network nodes), meaning that once data is written, it cannot be altered or deleted, preserving a clear and unchangeable transaction history.</p>

2.2.1.2 Core Components briefly described

Besides basic functional requirements, the NANCY Blockchain and its core components must be able to fulfil stakeholder requirements and processes as described in e.g. NANCY D4.1 and D6.2. Let us introduce now which core components are available for the NANCY stakeholders (mainly operators and users), and then in section 2.2.1 the reader will be introduced to their relation within the NANCY architecture (refer to D6.1).

The NANCY Blockchain

The NANCY Blockchain is a NEC-hosted, Hyperledger Fabric v2.2.0 based blockchain, with some security and privacy improvements.

Firstly, it implements the fastBFT consensus protocol, which is a fast and scalable BFT protocol. BFT protocols normally provide a guarantee of transaction finality and high security, but existing BFT protocols do not scale well. fastBFT, on the contrary, features a novel message aggregation technique that combines secure hardware with lightweight secret sharing. Combining this technique with several other optimizations (i.e., optimistic execution, tree topology and failure detection), fastBFT achieves low latency and high throughput even for large-scale networks.

Additionally, its scalability has been improved by implementing the work presented in <https://arxiv.org/pdf/2109.10302.pdf> (“MITOSIS: Practically Scaling Permissioned Blockchains”). Specifically, [21] it allows the dynamic creation of blockchains, as more participants join the system, to meet practical scalability requirements. Crucially, it enables the division of an existing blockchain (and its participants) into two—reminiscent of mitosis, the biological process of cell division. MITOSIS inherits the low latency of permissioned blockchains while preserving high throughput via parallel processing. Newly created chains are fully autonomous, can choose their own consensus protocol, and yet they can interact with each other to share information and assets—meeting high levels of

interoperability. MITOSIS can be ported with little modifications and manageable overhead to existing permissioned blockchains, such as Hyperledger Fabric.

In addition, a NANCY blockchain monitoring tool has been developed. For each channel, the application presents ledger-related data: Block and transaction counts, and network congestion indicators like throughput and mempool transaction count, accompanied by hourly and minute-based transaction and block graphs.

Lastly, privacy is one of the key requirements for NANCY. Digital privacy prevents the illegitimate use of users' personal data and automatically improves the reputation of the blockchain owner. The NANCY wallets feature different SSI methods that enable them to interact with an SSI infrastructure by possessing one or more verifiable credentials and generating verifiable presentations from them. Here, the "verifiable" objects are secured in a way that is *cryptographically verifiable*. These wallets become holders which feature "selective disclosure" (the ability of a holder to make fine-grained decisions about what information to share), which largely improves privacy in NANCY.

The NANCY wallets and blockchain adaptors

In the previous paragraph, we have mentioned the NANCY wallets. Section 3 is dedicated to them, however, we introduce some basic information for the sake of consistency.

The NANCY wallet is a Kotlin GRPC server which exposes calls for working with the NANCY blockchain, the PQC component and also the SSI infrastructure. The NANCY wallet (or *blockchain adaptor*, in certain contexts) also serves as a secure repository for the credentials and identities needed to access and interact with the Fabric-based NANCY blockchain network. In this sense, the NANCY wallet stores user identities, which can include (1) X.509 certificates as issued by the Fabric's Certificate Authority (CA) to authenticate a user or organization's identity, (2) the user's self-generated DIDs, and (3) private keys used to sign transactions on behalf of the user. It is important to repeat that the NANCY network is permissioned, so users must have valid credentials in their wallets to interact with the blockchain. Once authenticated, the user can e.g. query the ledger or submit transactions by interacting with the chaincode (e.g. the NANCY Marketplace).

Smart contract-based components

The fundamental smart contract-based component in NANCY is the NANCY Marketplace. By adding services in the Marketplace, the business support system (BSS) of a given operator can make them available to other domains. Consequently, any local operator, by means of the wallet of its service orchestrator, can search for available services in other domains that are able to fulfill an SLA that the local operator's capabilities cannot fulfill.

The Marketplace is integrated, through an oracle, with the Smart Pricing (SP) component to securely identify the best service in terms of price. It is also integrated, through a different oracle, with the Digital Agreement Creator (DAC) component to generate final agreements between remote operators (providers) and local operators (consumers), aimed at granting a certain service to a final user (UE, IoT, others).

The Marketplace is also integrated with the SLA Registry Smart Contract to manage the required signatures for said agreements.

Oracles are not smart contract-based components, but they are crucial components for enabling interaction between blockchains and the outside world. They are the bridge between the on-chain and off-chain data. Since blockchains are isolated by design, oracles act as intermediaries that provide

external information or trigger actions based on real-world events. In fact, oracles and blockchains work through *blockchain events*, which are signals emitted by a smart contract when specific conditions are met during a transaction. These are used to notify external applications – and oracles in this case. When the oracle listens to the blockchain event, it can fetch data from external sources (e.g., the Smart Pricing API), validate the authenticity and accuracy of the data, and inject this data into the blockchain through specific invokes.

Non-smart contract-based components

The two fundamental non-smart contract-based components that interact with the NANCY Blockchain are the Smart Pricing and the Digital Agreement Creator (DAC) components.

The Smart Pricing is an AI-driven system that adjusts prices dynamically in a digital marketplace, using learning algorithms and auction methods to ensure fair competition. It relies on a multi-agent system where providers compete in a blind reverse auction, with a built-in load-balancing feature that distributes clients efficiently while keeping prices competitive. The Smart Pricing securely delivers verified pricing data through a REST API, allowing smooth integration with the Marketplace.

The Digital Agreement Creator is an out-of-the-box software solution for creating Smart Contract code among NANCY stakeholders. Based on Java language and Spring Boot Framework, the DAC is fully dockerized. It can receive inputs via a RESTful interface concerning, e.g., providerId, consumerId, price, service requirements, etc. and then create ad-hoc containers which include the smart contracts code along with a unique identification number, generated by the DAC itself, and that works as a hash of the smart contract.

2.2.2 NANCY Blockchain Workflows and Relation to the NANCY Architecture

2.2.2.1 Relation to the Architecture

The revised and updated NANCY architecture was reported in D6.1 (see Figure 3). Two differentiated vertical domains are shown: The intra-operator domain (left) and the inter-operator domain (right). In addition to these, the architecture also proposes a set of distributed services to which the local service orchestrator can access.

Horizontal layers are divided into the infrastructure layer, the control layer, the orchestration layer and the business layer. Work in tasks T5.2 and T5.3 and, specifically, the NANCY Blockchain and its core components, belong to the business layer (horizontally) of the inter-operator domain (vertically).

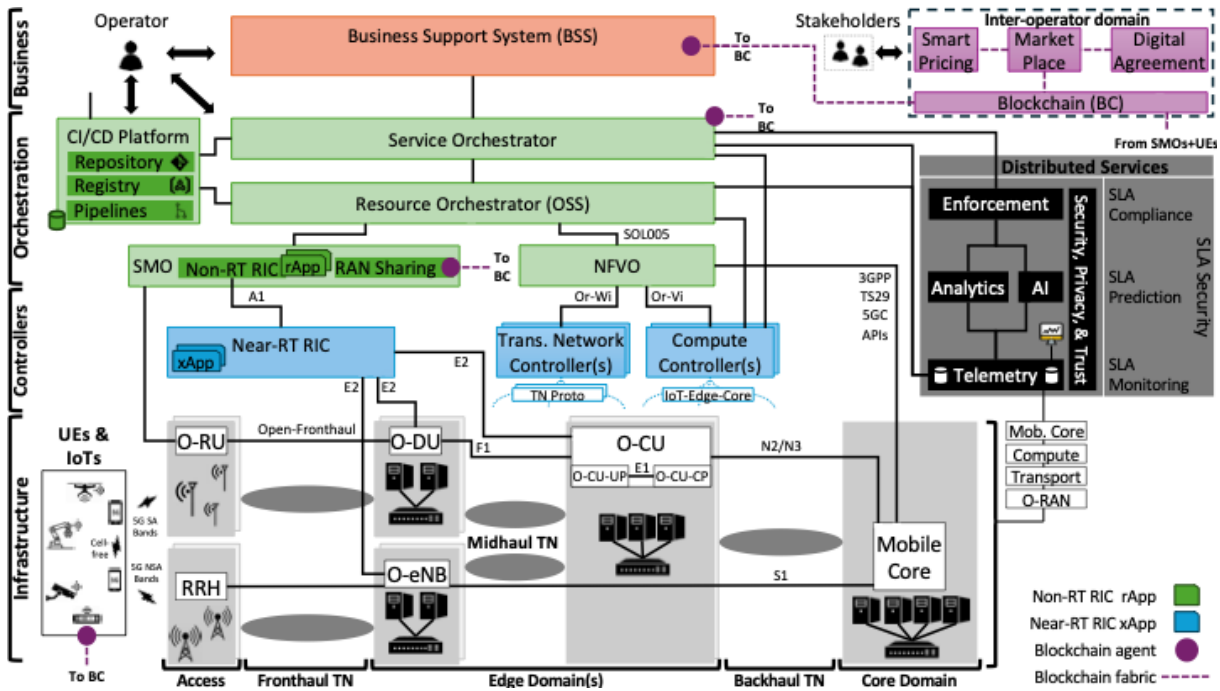


Figure 3. The updated NANCY architecture ⁵

However, in addition to the NANCY Blockchain and the core components residing in the inter-operator domain, wallets (for UEs and IoTs) and blockchain adaptors (analogous to wallets for less resource-constrained components) reside in the intra-operator domain (see Figure 4). These wallets and blockchain adaptors enable these components to securely and privately interact with the ledger.

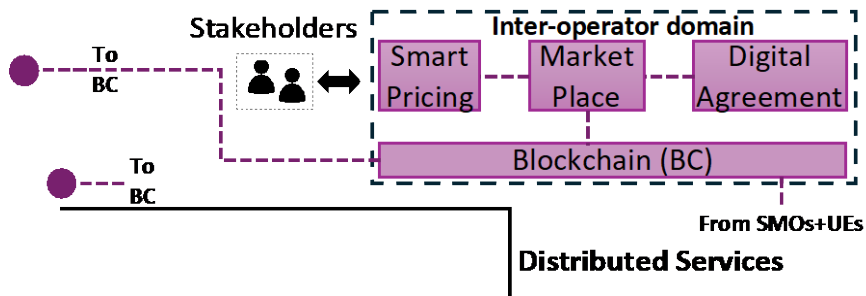


Figure 4. Detail of the inter-operator domain⁶

2.2.2.2 Basic service selection and agreement workflow (inside the inter-operator domain)

The following workflow was designed during P1 and presented in M18. The reader must note that the SSI capabilities of the NANCY wallet will be explained Section 3.

The workflow includes three fundamental stages: (1) listing services in the marketplace, (2) service selection and (3) SLA signature.

1- Listing services in the Marketplace

To interact with a Hyperledger Fabric blockchain using a wallet, the process first involves setting up and registering identities, obtaining credentials, and configuring the wallet for interaction. This is

⁵ Source: D6.1

⁶ Source: D6.1

common for all the entities wishing to communicate with the NANCY blockchain, but we will briefly explain it only here.

Every participant in the NANCY network must have a valid identity issued by the NANCY Certificate Authority (CA). This identity is stored in the wallet and used for authentication and transaction signing. For this to happen, firstly the network administrator uses their admin identity to register a new user with the CA. This step associates the new user with a specific role and organization. Then, the user uses the username and password to enroll with the CA. Enrollment generates the user's X.509 certificate and private key [22]. Now, the user can place its identity files (certificate and private key) in a structured directory, to then use the Fabric SDK (e.g., Node.js, Java, Go) to create and populate the wallet programmatically. Once the identity is in the wallet, the user connects to the Fabric network using a connection profile (YAML or JSON file). Once connected, the user can submit transactions or query the ledger through the smart contract API.

In our case, a dummy user called the *service provider* was equipped with a wallet and a valid identity. In addition, an initial version of the NANCY Marketplace was set up as a smart contract working in the NANCY Blockchain. Essentially, the Marketplace publishes a list of service providers together with services they can provide (with given performance indicators) plus minimum and maximum prices (for said services and indicators) plus the service providers' reputation (see Figure 5). The following API methods were designed and developed for the Marketplace:

Providers

- createProvider(JSON String)
- updateProvider(providerId String, JSON String)
- getProvider(providerId String)
- listProvider(JSON String)
- deleteProvider(providerId String)

Services

- createService(JSON String)
- updateService(serviceId String, JSON String)
- getService(serviceId String)
- listService(JSON String)
- deleteService(serviceId String)

Searches

- createSearch(JSON string)
- getSearch(searchID String)
- listSearch(JSON string)
- setSearchPricing(searchID string, JSON string)
- setSearchSLA(searchID string, JSON string)

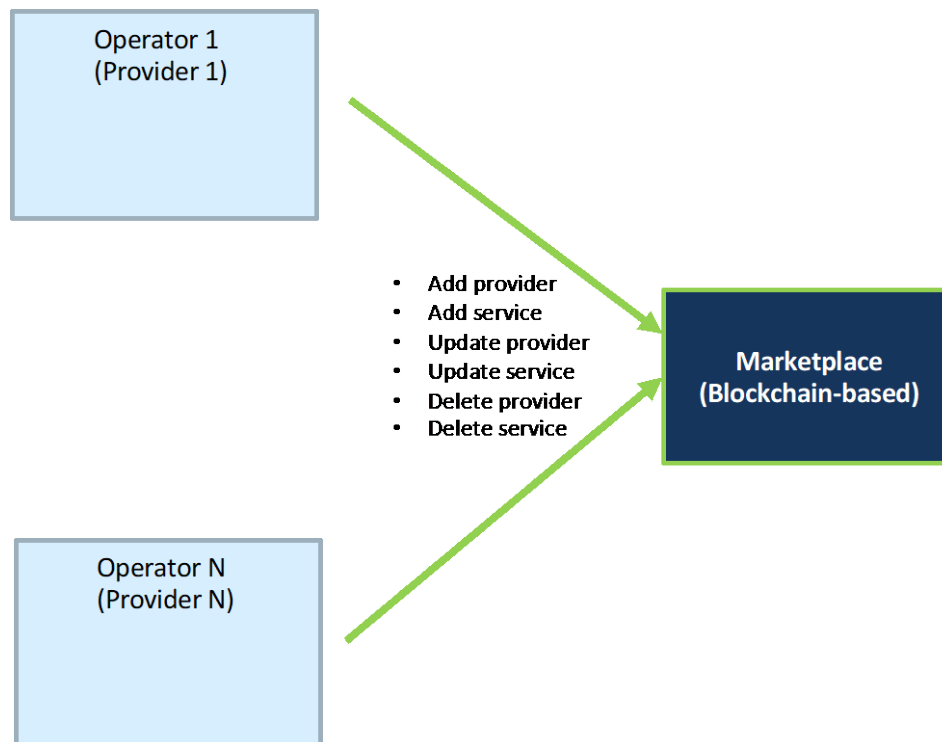


Figure 5. Listing services in the marketplace

With this setup, every service provider is able to add, update or delete itself as a provider and to add, update or delete services using GRPC calls to the Marketplace smart contract, effectively changing the ledger status. For each service, several key performance indicators (KPIs) are included, like throughput or response time. These indicators can be considered for later service searches, so this will make possible the service selection in stage 2. In addition, minimum and maximum prices for each service are also added – these will later be fed to the Smart Pricing component. Finally, each service provider starts listening to events coming from the SLA Registry smart contract (see next paragraph), which will make possible the SLA signature in stage 3.

2- Service selection

The basic service selection consists of several steps (1-9 in Figure 6) and involves different on-chain and off-chain processes. Here it is worth noting that at M18 the system did not care who the service consumer and service provider were – inside the NANCY Architecture. The system was designed to work on its own and demonstrate how the Marketplace, Smart Pricing and Digital Agreement Creator could handle different events and data. Two smart contracts exist in this workflow: the Marketplace (for Stage 2) and the SLA Registry smart contract (for Stage 3), where SLAs are registered and verified.

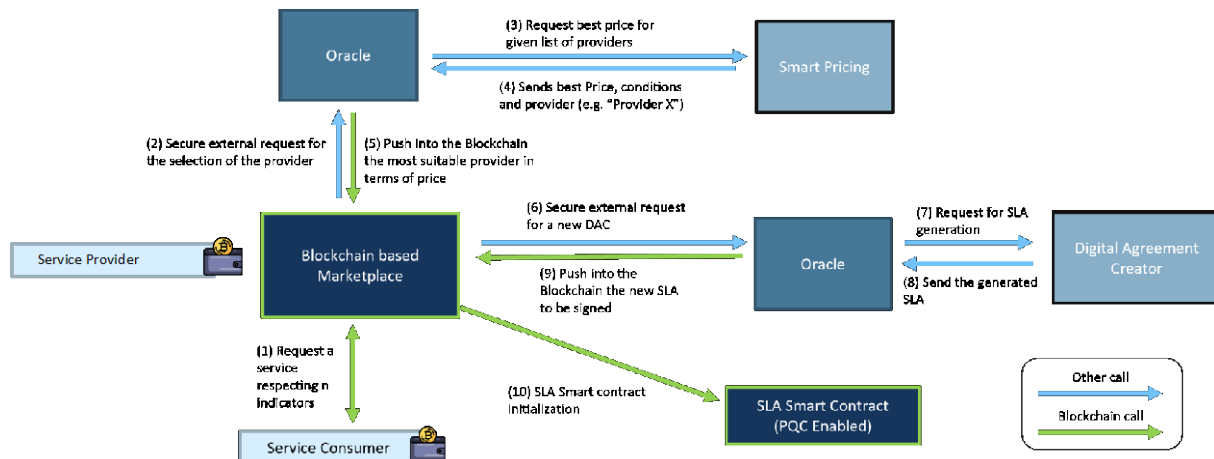


Figure 6. Service selection

We assume that all components have been successfully registered and that stage 1 (listing services in the Marketplace) has also been successful; thus, the NANCY Marketplace lists services from different providers. Stage 2 begins with a *service consumer* interacting with the Marketplace chaincode by means of its NANCY wallet gateway (step 1). If the service consumer is a user equipment, then (see Section 3.2) its wallet is equipped with PQC capabilities (for Stage 3) and must register its public key on chain. Said interaction is done through GRPC commands, specifically searching for a service that satisfies certain requirements (continuing with the explanation above, these could include throughput, response time and maximum price). In addition to this, after calling this search, the service consumer wallet subscribes (starts listening) to blockchain events triggered by the SLA Registry smart contract. The marketplace searches its stored information to identify those registered services whose parameters meet the requirements set out in the request.

Step 2 would see the Marketplace smart contract building a response to the search method, and triggering a blockchain event, which the first oracle would listen to. This is done since the Smart Pricing component is off-chain, meaning it is not a smart contract and thus cannot be directly called by another smart contract (the Marketplace, in this case). Step 3 is an HTTPS request to the Smart Pricing component, but this time only containing the services that the Marketplace considered adequate for the service consumer's search – this is, services that met the service consumer's original criteria. After the Smart Pricing component (see D4.5 and D5.2's section 2.3.2) produces its debating processes and makes its calculations, it is ready to send back, in step 4, the selected service and its conditions to the first oracle, which in turn (see 2.3.2.3) processes this response to create the suitable transaction to communicate it back to the Marketplace smart contract (step 5).

Step 6 would see the Marketplace smart contract triggering a blockchain event, which the second oracle would listen to. This is done since the Digital Agreement Creator component is off-chain, meaning it is not a smart contract and thus cannot be directly called by another smart contract (the Marketplace, in this case). Step 7 is an HTTPS request to the Digital Agreement Creator component, containing specific data about the service that the Smart Pricing considered the best candidate for the service consumer's original search, in terms of requirements, and also price. Then, the Digital Agreement Creator is able to turn these data into an SLA that should satisfy both parties: The service provider and the service consumer. This SLA is pushed back into the second oracle (step 8), which processes it to create the suitable transaction to send it back to the Marketplace smart contract in step 9. Then, the Marketplace smart contract calls the SLA Registry smart contract, which registers the SLA on-chain (step 10). This concludes stage 2.

3- SLA Signature

As introduced in stages 1 and 2, both the service providers and the service consumer are subscribed (listening) to blockchain events triggered by the SLA Registry smart contract.

If provider X is not selected as the best match, nothing happens; but if provider Y is indeed selected as the best match by the Smart Pricing component, it will receive – after the Digital Agreement Creator builds the final SLA – an event for SLA creation. This will also happen for the service consumer, and both provider Y and service consumer will be asked if they wish to sign the final SLA.

When the provider proceeds, its wallet issues a signing transaction to the blockchain. The transaction ID serves as a valid signature in the SLA. In the case of the service consumer, if it is a user equipment, its wallet will issue a transaction with a PQC signature, which can be automatically verified by the SLA Registry smart contract. Finally, both parties are informed of the signed SLA since they are subscribed to such events.

2.2.2.3 Extended Service Selection and Agreement Workflow (combining the intra and inter-operator domains)

The following workflow was designed during P2 and presented in M24. The main difference between this and the basic service selection and agreement workflow is that now the inter-operator domain works together with various WP3 and WP4 actors in the intra-operator domain.

The workflow includes three fundamental stages: (1) listing services in the marketplace, (2) service selection inside the offloading and caching workflow and (3) SLA signature.

1- Listing services in the Marketplace

Listing services in the Marketplace does not change from the basic workflow (see Figure 7). Once connected to a blockchain (see description of the basic workflow above), any user can submit transactions or query the ledger through the smart contracts APIs. The main difference is that the originally called “dummy user” and identified as the *service provider* is now the Business Support System (BSS) of a local operator. Hence, it is this BSS the one equipped with a wallet and a valid identity. The NANCY Marketplace is set up as a smart contract working in the NANCY Blockchain, just as in the basic workflow. Now, the first step is that all the operators, for each local domain, use the Marketplace to publish a list of services they can provide, with performance indicators (service requirements), other data (e.g. computing resources, localization data⁷) plus minimum and maximum prices, plus the operator’s reputation and ID data.

There is no need for extension to the API methods designed and developed for the Marketplace in the basic workflow, but the reader is referred to 2.3.2 for more details. With this set of methods, every service provider (local operator) is able to add, update or delete itself as a provider and to add, update or delete services using GRPC calls to the Marketplace smart contract, effectively changing the ledger status. For each service, several indicators are included, like throughput or response time. These indicators can be considered for later service searches coming from other domains. This will make possible the service selection in stage 2. In addition, minimum and maximum prices for each service are also added – these will later be fed to the Smart Pricing component. Finally, we assume that at this stage, each service consumer (final client) and service provider (local operator) are listening to events

⁷ Fields for computing resources and localization data are being implemented as part of WP6

coming from the SLA Registry smart contract in the inter-operator domain, which will make possible the SLA signature in stage 3.

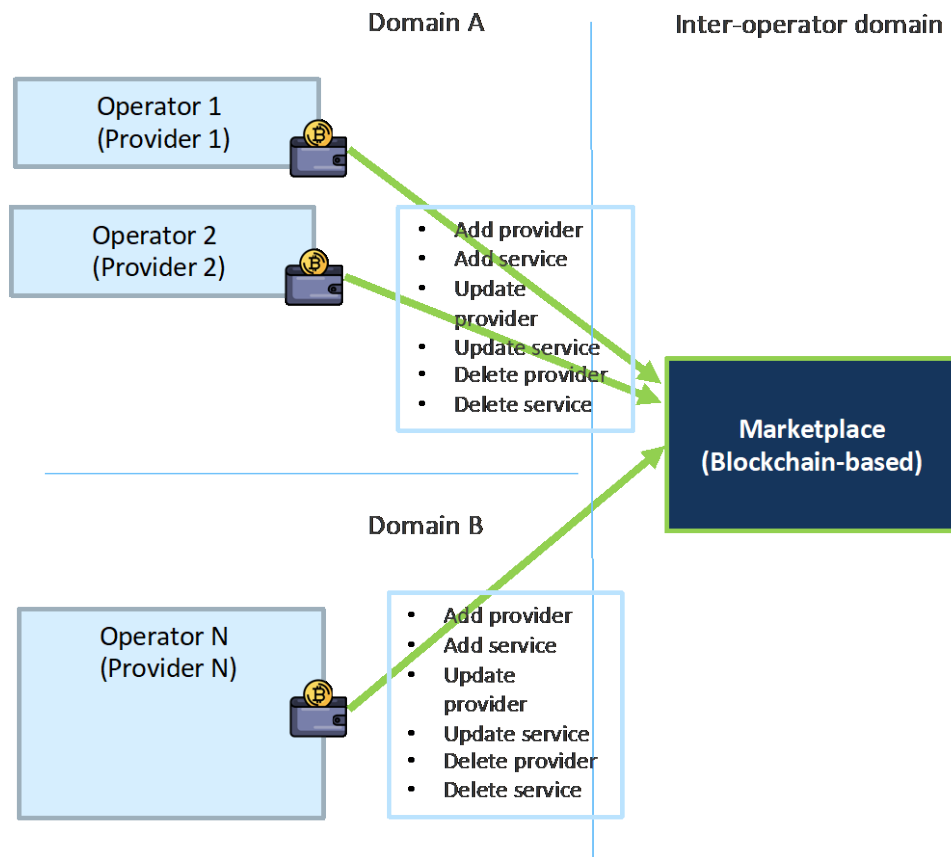


Figure 7. Listing services in the marketplace

2- Service selection inside the offloading and caching workflow

The basic service selection consisted of several steps (1-9 in Figure 6) and involved different on-chain and off-chain processes, but the system did not care who the service consumer and service provider were inside the NANCY Architecture. This is different in the extended workflow, as anticipated in Figure 8.

We assume that all components have been successfully registered and that stage 1 (listing services in the Marketplace) has been also successful; thus, the NANCY Marketplace lists services from different operators in different domains.

We also assume that a service request *from a client* ("Client 1") has been received and analyzed by the BSS of its local operator (Operator 1). This BSS has created an initial SLA and sent it to the local SO (of Operator 1). Since – we assume – the local SO is not able to handle the request, a notification is sent back to the local BSS, which forwards the request to the inter-operator domain.

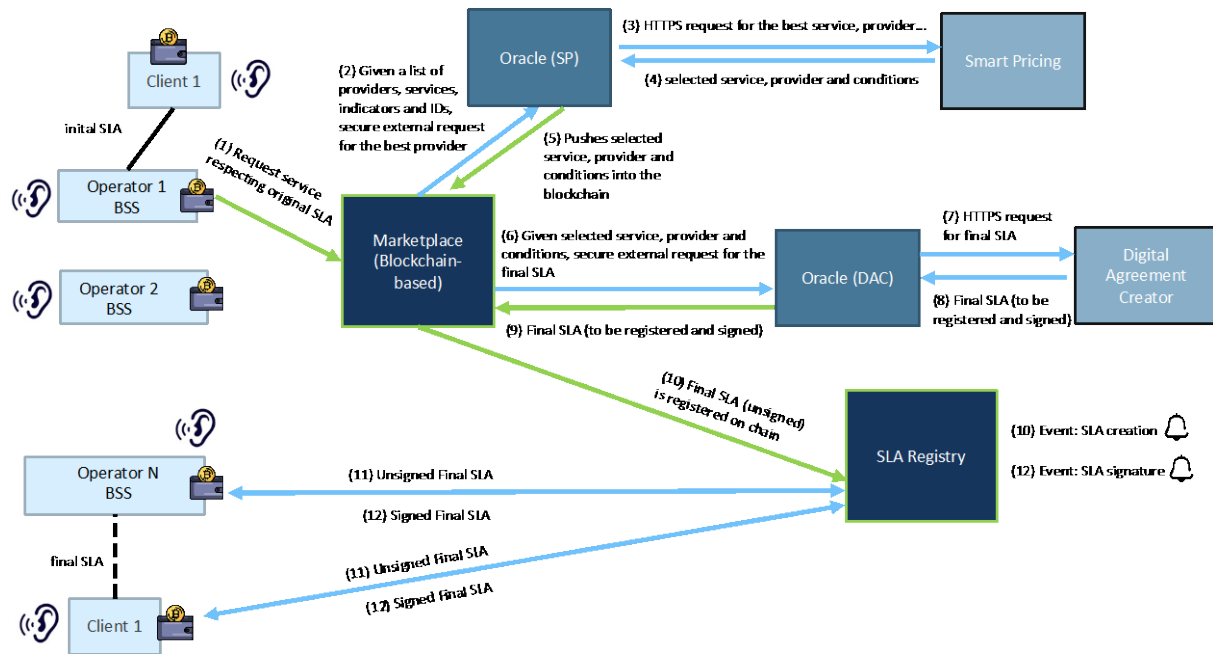


Figure 8. Service selection inside the offloading and caching workflow

Stage 2 thus begins with the BSS (analogous to the *service consumer* in the basic workflow) of a local operator interacting with the Marketplace chaincode by means of its NANCY wallet gateway (step 1) through GRPC commands, specifically searching in the inter-operator domain for an operator able to host the service request that satisfies the initial SLA that the local operator cannot fulfil in the intra-operator domain. We will call this local operator “Operator 1”. Operator 1’s BSS, in the same step, uses its wallet to subscribe (start listening) to blockchain events triggered by the SLA Registry smart contract inside the inter-operator domain. Operator 1 will not sign the new SLA, but it must remain informed when such new SLA – between a remote operator and Operator 1’s client – is signed. This information can then be used for e.g. billing between operators, or SLA compliance.

Step 2 is similar to that of the basic workflow, and it would see the Marketplace smart contract building a response to the search method which Operator 1’s BSS called. This response contains a list of services from different operators in different domains that are able to satisfy the initial SLA. In addition to this, the Marketplace triggers a blockchain event, which the first oracle would listen to. As explained, the Smart Pricing component is off-chain, meaning it is not a smart contract and thus cannot be directly called by another smart contract (the Marketplace, in this case). Step 3 is an HTTPS request to the Smart Pricing component, only containing the services that the Marketplace considered adequate for fulfilling the initial SLA. After the Smart Pricing component (see D4.5 and D5.2’s section 2.3.2) produces its debating processes and makes its calculations, it is ready to send back, in step 4, the selected service, provider and conditions to the first oracle, which in turn will verify this response before communicating it back to the Marketplace smart contract (step 5).

Step 6 would see the Marketplace smart contract triggering a new blockchain event, which the second oracle would listen to. Again, this is done since the Digital Agreement Creator component is off-chain, meaning it is not a smart contract and thus cannot be directly called by another smart contract. Step 7 is an HTTPS request to the Digital Agreement Creator component, containing specific data about the original client, the new service provider (new operator) and the service that the Smart Pricing considered the best candidate for the original search, in terms of requirements and also price. Then, the Digital Agreement Creator is able to turn these data into a *new* SLA that should satisfy both parties:

the new service provider (let us call it Operator N) and the service consumer (Operator 1's original client). Note that, even though it was Operator 1's BSS the one who triggered stage 2, it is not signing the new SLA. This new SLA is pushed back into the second oracle (step 8), which processes it creates the suitable transaction to send it back to the Marketplace smart contract in step 9. Then, the Marketplace smart contract calls the SLA Registry smart contract, which registers the SLA on-chain (step 10). This concludes stage 2.

3- SLA Signature

Two events can be triggered by the SLA Registry smart contract: "SLA Creation" and "SLA Signature". As introduced in stages 1 and 2, both the service provider (Operator N, but actually, any operator that has listed services in the Marketplace) and the original searcher (Operator 1) are subscribed (listening) to these blockchain events. However, also the *final* service consumer – Client 1; this is, Operator 1's client – is listening for blockchain events triggered by the SLA Registry smart contract

For any operator different to "Operator 1" or "Operator N" and for any client different to "Client 1", nothing will happen after the SLA Registry "SLA creation" event is triggered. But both Operator N and Client 1 will be asked if they wish to sign the final SLA – as built by the DAC component. The signatures take place similarly to those explained for the basic workflow. Finally, both parties are informed of the fully signed SLA since they are subscribed to the "SLA signature" event of the SLA Registry smart contract, but in addition to them, also Operator 1 receives such event and can act accordingly.

2.2.2.4 About Privacy with Blockchain Smart Contracts

In Hyperledger Fabric, privacy and access control can be applied in different granularity. This is especially important in the case of NANCY for what pertains to the SLA private data between providers and consumers (see SLA model defined in D4.1).

First, the Fabric blockchain runs multiple channels, each configured (via `configtx.yaml`) with a consortium composed of the organizations that are allowed to participate in the channel communication. Only nodes presenting certificates that are issued by the CAs from these organizations can see ordered transactions or emitted events in that channel.

Secondly, each component in the blockchain can specify their access control policies [23]. More specifically, each organization, orderer service, channel can define their customized policies for different operations such as transaction READ, transaction WRITE, transaction endorsement, and administrator.

Moreover, inside each smart contract, the smart contract developers can further define and enforce access control policies by checking the IDs (i.e., certificates) of the transaction issuer against the state in the ledger. What's more, Hyperledger Fabric boasts an additional private data [24] mechanism, that further restricts the visibility of the data in the ledger. This is, private data is only visible as plaintext in the ledger hosted on the peers from authorized organizations. For other peers in the same channel, only the hash of the private data is saved in the ledger. And the private data is presented only by its hash in the block so that its content is also kept secret from the ordering service (i.e., the consensus protocol).

In what follows, we describe the privacy measures deployed in the NANCY blockchain to ensure different data privacy (i.e., not limited but including SLA data). Some of them have been implemented in the PoC of the workflow.

1. *Organization CAs and TLS.* All transactions must provide a valid certificate issued by a CA from an organization defined in a consortium presented in any channel, and all communication is secured with TLS.
2. *Channels.* The marketplace and SLA Registry smart contracts are deployed in a channel composed of the consortium restricted to NANCY partners only. SSI-related smart contracts, however, are deployed in a channel that is accessible by a much wider public, as all information is public and anonymous.
3. *Privacy of the SLA data.* We can apply the private data mechanism on the SLA data. In this way, only organizations (industrial operators) involved in the SLA contracts can see its content while the others only witness the hash of the SLA for timestamping purpose. Moreover, since individual consumers, unlike the providers, are defined as from the same organization, we do not authorize them access to the private data. Rather, we attach an extra copy of the encrypted SLA using the public key of the individual consumer, which is retrievable from the DID registry, so that the consumer can decrypt the SLA and match the hash value that is publicly visible in the channel. Similar privacy is also enforced in the SLA events (see option *DeliverWirthPrivateData* in [25]). Regarding the inter-operator domain scenario described previously, where a second operator is involved in the service searching process, the same solution applies. More specifically, Operator 1 searches a service on behalf of Consumer 1 and the marketplace found Operator N as the best match creates an SLA between Operator N and Consumer 1. We set SLA as private data between Operator 1 and Operator N, while attaching the encrypted SLA for Consumer 1. As a result, *all three and only these three entities can see the content of the SLA.*
4. *Access control to SLA creation and signing.* SLARegistry smart contract checks the certificates of the transaction senders. SLA creation is only allowed if the sender presents a certificate of the NANCY oracle, and SLA signing is only allowed if the sender is either the provider or the consumer of the SLA in question.
5. *Access control to DIDRegistry and VRegistry.* For write operations such as UPDATE and DELETE, DIDRegistry checks the certificates of the transaction senders and permits the operation only when the transaction sender presents the certificate corresponds to the DID in question. Similarly, the VRegistry checks that the sender certificate must correspond to the issuer's DID in question.

2.3 The NANCY Blockchain Core Components

2.3.1 Smart Contract-based Components

2.3.1.1 SLA Registry

Description

We deployed a smart contract to maintain all the SLA contracts. This smart contract allows corresponding partners to create, search and sign the SLA contract. The SLA contract data structure is defined as follows:

```
type SLA struct {
    Id      string `bson:"id"      json:"id"`
    Value   string `bson:"value"   json:"value"`
    ProviderId string `bson:"provider_id" json:"provider_id"`
    ConsumerId string `bson:"consumer_id" json:"consumer_id"`
    ProviderSig string `bson:"provider_sig" json:"provider_sig"`
    ConsumerSig string `bson:"consumer_sig" json:"consumer_sig"`
}
```

- *Id*- unique SLA contract ID.
- *Value* – JSON string of the SLA contract content. This can be flexibly defined by the contract creator.
- *ProviderId* – DID of the provider involved in the SLA contract.
- *ConsumerId* – DID of the consumer involved in the SLA contract.
- *ProviderSig* – transaction ID used as the signature of the provider, since the transaction is already signed by the provider using his blockchain credential.
- *ConsumerSig* – PQC signature of the consumer encoded in HEX string.

Interfaces

The smart contract further defines the following interfaces as described in Table 2. It will emit event *InitSLA* and *SigningSLA* for subscription.

Table 2. Interface description of smart contract SLA Registry

Chaincode Interface Description
InitSLA (id <i>string</i> , value <i>string</i> , providerId <i>string</i> , consumerId <i>string</i>)
creates a new SLA entry in the ledger using the data structure described above and indexed by the <i>id</i> , which stands for the SLA unique ID. Access control policy is enforced to check whether the ID of the transaction issuer is “NancyOracle”. At the end, it emits an event <i>InitSLA</i> .
GetSLA (slaId <i>string</i>)
returns the SLA entry given the <i>slaId</i> . If the corresponding SLA does not exist, returns error.
GetSLAByConsumerId (consumerId <i>string</i>)

returns the list of SLA entries associated with the given `consumerId`. If the corresponding SLA does not exist, returns error.

SignSLA(`slaId string`)

signs the SLA corresponding to the provided `slaId` without PQC. Access control policy is enforced to check whether the transaction issuer presents the same ID as the provider or the consumer in the SLA in question. The function further checks whether the role of the transaction issuer is *not* “UE”. Note that the ID and the role of the transaction issuer can be extracted from the sender’s ECert and the transaction ID is saved as the signature.

Once all the checks have passed, the transaction ID is saved in the `providerSig` or `consumerSig` field. At the end, it emits an event *SigningSLA*.

SignSLAPQC(`slaId string, sigStr string`)

signs the SLA corresponding to the provided `slaId` with PQC. Access control policy is enforced to check whether the transaction issuer presents the same ID as the provider or the consumer in the SLA in question. The function further checks whether the role of the transaction issuer is “UE”. Then it looks up the PQC public key indexed by the same transaction issuer from the **didRegistry** chaincode in order to verify the PQC signature `sigStr`, which covers the `value` field. Note that the provided `sigStr` must be encoded in Hex string.

Once all the checks have passed, the `sigStr` is saved in either the `providerSig` or `consumerSig` field. At the end, it emits an event *SigningSLA*.

2.3.1.2 Marketplace

Description

The NANCY marketplace is a business-layer chaincode where operators can engage in the exchange of services to keep their quality of service for their customers. The idea is to allow the interdomain flow described in section 2.2.2.3 providing a tool for operators to register (through their BSS) their available services as well as to consume and offload available services from other operators when needed.

Figure 9 shows key interactions of the marketplace. Since the marketplace is chaincode residing on the blockchain, any user interacting with the marketplace will use a wallet. This refers to:

- Operators providing information about themselves (Provider Endpoints) and their services (Service Endpoints) or making requests for searching other available services (Search Endpoints).
- Oracles. Additionally, the marketplace interacts with oracles through APIs, while the oracles, which also have a wallet, interact with the marketplace to provide the most suitable service for a request based on the best price (Smart Pricing Endpoint) and the created digital agreements from a search (Digital Agreement Creator Endpoint). Moreover, the marketplace also interacts with the SLA Registry Smart contract (see section 2.3.1.1) through another oracle, which directly requests the contract to initiate the digital agreement signature process between the client and the new operator. This oracle is needed because the call from a chaincode (Marketplace Smart Contract) to another chaincode (SLA Registry Smart Contract) does not allow the emission of events.

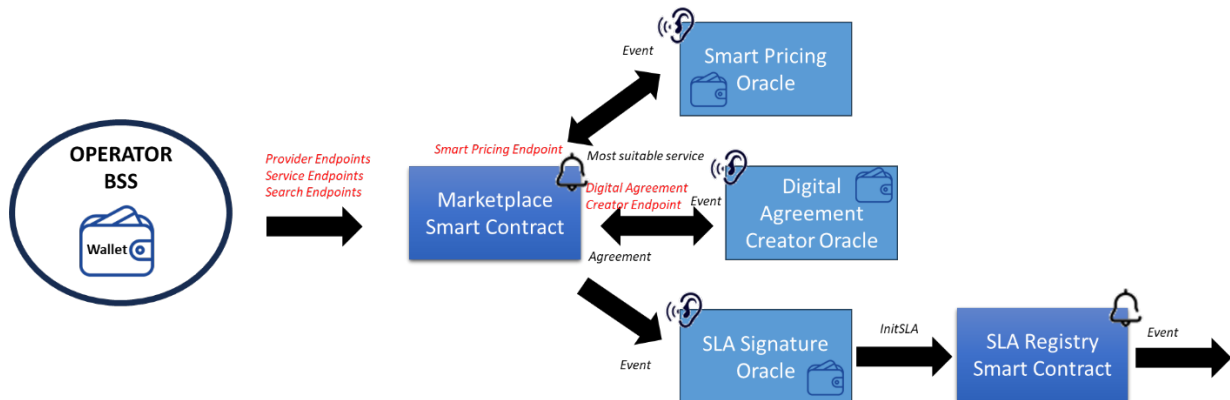


Figure 9. Marketplace main interactions

The marketplace has been designed in a general and adaptable way, so that information of different data models can be stored. For this reason, there is a base model structure defined as follows:

```

{
  "model_type": "nancy_provider",
  "model_version": "0.1.2",
  "owner": "54f23c05b8c66125c870cd0c89bf7c08978fcc28c0cce126be78c5ad0887ef78",
  "mspi_id": "org0-nancy-dev",
  "timestamp": 1725538763,
}

```

- **model_type**: This refers to the data model for registering information. Three different data models have been considered in the marketplace: [nancy_provider, nancy_service, nancy_agreement]. Type: string.
- **version**: This refers to the version of the specific data model; currently in version 0.1.2, but it can be updated for dealing with new data models without needing to remove previous recorded information. Type: string.
- **owner**: This refers to the id of user who creates the data. Type: string.
- **mspi_id**: This refers to the mspi_id of user who creates the data. Type: string.
- **timestamp**: This refers to the creation timestamp. Type: timestamp

As mentioned, there are three data models in the marketplace:

NANCY Provider

The current data model for NANCY providers, which refers to the information associated to a specific operator, is:

- **id**: Operator identifier. Type string.
- **name**: Name of the provider. Type: string.

- **type:** Type of provider. Type: string.

This data model could be updated to include additional information related to the operator (i.e., resources availability) as needed.

NANCY Service

The data model for NANCY services, which refers to the information associated to the services offered by the operators in the inter-operator domain and the features required for the SLA according to D4.1, is:

- **providerID:** The ID of the operator the service belongs to. Type string
- **minPrice:** Minimum price. Type float64
- **maxPrice:** Maximum price. Type float64
- **Duration:** Duration. Type string
- **ResponseTime:** Response time. Type string
- **Throughput:** Throughput. Type string
- **Latency:** Latency. Type string

This data model could be updated to include additional information about services (i.e., location) based on the SLA requirements.

NANCY Search

The data model for NANCY search, which refers to the information management for the search of a new operator in the inter-operator domain, is:

- **consumer_ID:** The ID of the user making the request. Type string.
- **status:** The status of the search. Type string. Different types of status:
 - INIT: Initial state.
 - PRICE: Price state. Ready to set price through Smart Pricing Component.
 - SLA: SLA state. Ready to set SLA through DAC Creator Component.
 - FINISHED: Search is finished.
 - Additionally, ERROR refers to an unrecoverable error occurred.
- **services:** The suitable services fulfilling the request filters. Type List<JSON>
- **pricing:** The most suitable service according to the Smart Pricing Component. Type JSON.
- **sla:** The created filled SLA. Type JSON.

This data model could be updated to include additional information if needed.

As mentioned, the search has different status as it requires the operation of different NANCY components (marketplace itself, Smart Pricing and DAC Creator) as explained in Figure 10. As a result, a notification to the SLA oracle is sent to initialize the SLA signature process.

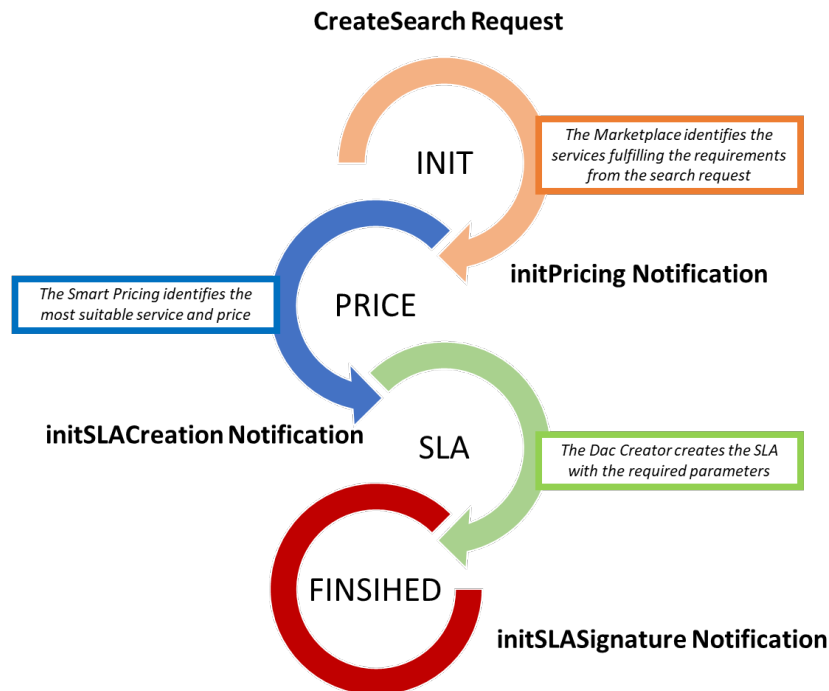


Figure 10. Marketplace search status

Interfaces

The smart contract further defines the following interfaces as described in Table 3 for the different endpoints shown in red in Figure 9.

Table 3. Interface description of smart contract Marketplace

Provider Endpoints Chaincode Interface Description
createProvider (id <i>string</i> , name <i>string</i> , type <i>string</i>)
Invoke (Write operation). It allows the registration of a new operator in NANCY marketplace providing the information included in the NANCY Provider data model. It returns the identifier of the registered operator inside the marketplace.
updateProvider (id <i>string</i> , data <i>string</i>)
Invoke (Write operation). It allows the information update about an already registered operator in NANCY marketplace. The "id" identifies the operator identifier to be updated while "data" refers to a JSON with information to be updated (i.e., "name" : "orange"). It returns the identifier of the updated operator inside the marketplace.
getProvider (id <i>string</i>)
Query (Read operation). It allows to get the registered information about a specific operator indicated in "id". It returns the registered details, including id, name and type.
listProvider (id <i>string</i>)
Query (Read operation). It allows to list the complete list of registered operators in the marketplace. "id": { "\$regex": ".*" }, is the way to list all the registered operators. As a result, an

array with all the registered operators information is obtained. NOTE: this functionality has been implemented in such a general way that filters could be applied to the list (i.e., "id": { "type": "operator" }).

deleteProvider (id *string*)

Invoke (Write operation). It allows to unregister an operator identified by "id" from the marketplace. It returns the identifier of the removed operator inside the marketplace.

Service Endpoints Chaincode Interface Description

createService (provider_id *string*, minPrice *float64*, maxPrice *float64*, duration *string*, responseTime *string*, throughput *string*, latency *string*)

Invoke (Write operation). It allows the registration of a new service in NANCY marketplace providing the information included in the NANCY Service data model. For example:

- "provider_id": it refers to the operator identifier offering this service (i.e., e07ba9cf13653760fd98e3b9553a94a49aa6cb445f4be10789810e2115a1de65).
- "minPrice": it refers to the minimum price for the service to be considered by the smart pricing component (more details in WP4). (i.e., 35.00).
- "maxPrice": it refers to the maximum price for the service to be considered by the smart pricing component (more details in WP4). (i.e., 125.00).
- "duration": it refers to the time this service is available (in minutes) (i.e., 60).
- "responseTime": it refers to the response time of the service (in seconds) (i.e., 30).
- "throughput": it refers to the throughput of the service (in Mbps) (i.e., 50).
- "latency": it refers to the latency of the service (in milliseconds) (i.e., 3).

It returns the identifier of the registered service inside the marketplace.

updateService (id *string*, data *string*)

Invoke (Write operation). It allows the information update about an already registered service in NANCY marketplace. The "id" identifies the service to be updated while "data" refers to a JSON with information to be updated (i.e., "latency" : "33"). It returns the identifier of the updated service inside the marketplace.

getService (id *string*)

Query (Read operation). It allows to get the registered information about a specific service indicated in "id". It returns the registered details, including id, provider_id, minPrice, maxPrice, duration, responseTime, throughput, latency.

listService (id *string*)

Query (Read operation). It allows to list the complete list of registered services in the marketplace. "id": { "\$regex": ".*" }, is the way to list all the registered services. As a result, an array with all the registered services information is obtained. NOTE: this functionality has been implemented in such a general way that filters could be applied to the list (i.e., "id": { "provider_id": "XYZ" }).

deleteService (id *string*)

Invoke (Write operation). It allows to unregister a service identified by “id” from the marketplace. It returns the identifier of the removed service inside the marketplace.

Search Endpoints Chaincode Interface Description**createSearch (consumer_id *string*, servicequery *string*)**

Invoke (Write operation). It allows the generation of a new search of suitable services in the marketplace. Internally, communications with Smart Pricing oracle, Digital Agreements Creator oracle and SLA oracle happens. The parameters are as follows:

- "consumer_id": it refers to the operator identifier making the services search request (i.e., e07ba9cf13653760fd98e3b9553a94a49aa6cb445f4be10789810e2115a1de65).
- "servicequery": it refers to the minimum requirements for the required services (i.e., "latency": "32").

It returns the identifier of the current search.

During its execution, as explained in Figure 10, different events are emitted to notify the oracles to start with their operation:

- initPricing event, to notify the Smart Pricing oracle to start with the Smart Pricing operation.
- initSLACreation event, to notify the Digital Agreement Creator oracle to start with the SLA creation process.
- initSLASignature event, to notify the SLA Signature oracle to start with the SLA signature process (managed by the Sla Registry smart contract).

getSearch (id *string*)

Query (Read operation). It allows to obtain the status and result of a requested search indicated in “id”. It returns the services (from PRICE status), pricing (from SLA status) and sla information (from FINISHED status).

listSearch (id *string*)

Query (Read operation). It allows to list the complete list of searches done in the marketplace. "id": { "\$regex": ".*" }, is the way to list all the searches. As a result, an array with all the search details is obtained.

deleteSearch(id *string*)

Invoke (Write operation). It allows to remove (stop) a search identified by “id” from the marketplace. It returns the identifier of the removed search inside the marketplace.

Smart Pricing Endpoint Chaincode Interface Description**setSearchPricing (id *string*, data *string*)**

Invoke (Write operation). It allows the Smart Pricing Oracle to notify the marketplace the most suitable service and its price for the search identified by “id”. “data” contains the service identifier and its suitable price (i.e., "provider_id": "providerX", "service_id": "service123", "price": 71.3093472).

Digital Agreement Creator Endpoint Chaincode Interface Description

```
setSearchSLA (id string, data string)
```

Invoke (Write operation). It allows the Digital Agreement Creator Oracle to notify the marketplace created SLA for the search identified by “id”. “data” refers to the information inside the created digital agreement (i.e., "id": "daclD", "consumer_id": "consumer_id", "provider_id": "providerX", "service_id": "service123", "price": 10000, "service_description": [], "hash_smartcontract": "0x123").

2.3.1.3 Smart Contracts related to SSI

Description

We further defined the DIDRegistry and the VCRegistry smart contracts to enable SSI-compatible solutions. These smart contracts turn the blockchain to a public data registry for any party to look up ways to authenticate or verify any identities in a privacy-preserving manner.

We ask readers to refer to Section 3.1.1 for a detailed description of these smart contracts together with their interfaces.

2.3.2 Oracles and Non-Smart Contract-based Components

2.3.2.1 Digital Agreement Creator

Description

The Digital Agreement Creator (DAC) is an out-of-the-box software solution for creating Smart Contract code among NANCY stakeholders. Based on Java language and Spring Boot Framework, the DAC is fully dockerized. It can receive inputs via its RESTful interface concerning e.g., providerId, consumerId, service, price, conditions, etc. and then create ad-hoc containers that hold the smart contracts for relevant parties based on the provided input along a unique identification number, generated by the DAC itself, and that works as a hash of the smart contract.

The application dynamically generates and manages these ad-hoc containers-holders of the smart contracts for the Hyperledger Fabric blockchain via REST API requests. Its scalable design and modular architecture separate concerns between orchestration, generation, and API layers while its RESTful APIs enable extensibility and integration with other systems.

Architecture

Figure 11 represents the high-level architecture of the DAC component.

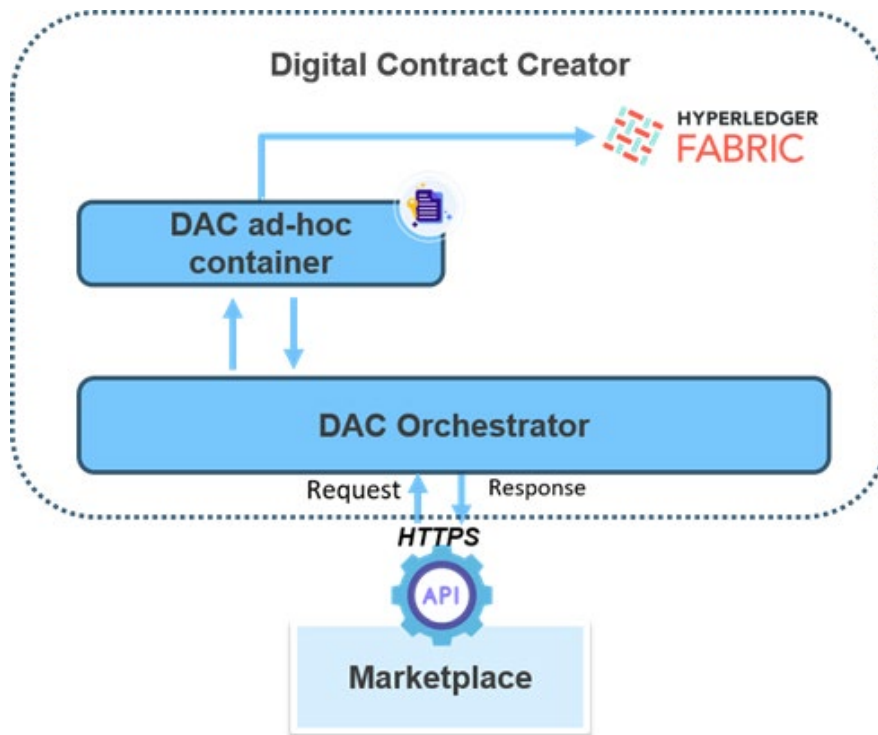


Figure 11. The high-level architecture of DAC component.

Interfaces

The DAC provides RESTful interfaces to manage ERC721-like tokens on a Hyperledger Fabric network. It includes the following key endpoints:

Create SmartContract based on a request

A REST API POST method (/DAC/createSLA) that creates a Chaincode for FabricLedger based on the request and returns the ID and Hash of the Smart Contract.

Example of the request in JSON format:

```

{
  "provider_id": "c9d1976354f9f069a04600cbb0456713401addc658bd46e50ca6a56392c04a32",
  "consumer_id": "consumer1",
  "price": 72.3444554,
  "service_id": "servieNomehacescaso",
  "service_description": [
    {
      "provider_id": "c9d1976354f9f069a04600cbb0456713401addc658bd46e50ca6a56392c04a32",
      "minprice": 30,
      "maxprice": 115,
    }
  ]
}
  
```

```
"duration": "60",
"responseTime": "30",
"throughput": "12",
"latency": "32"
},
{
"provider_id": "c9d1976354f9f069a04600cbb0456713401addc658bd46e50ca6a56392c04a32",
"minprice": 35,
"maxprice": 120,
"duration": "62",
"responseTime": "32",
"throughput": "15",
"latency": "32"
}
]
}
```

Get SmartContract by its hash

A REST API GET method (/DAC/getSmartContract/{hash}) that gets (downloads) the SmartContract based on its Hash.

These interfaces are designed to simplify interactions with the blockchain, enabling developers to integrate token minting and ownership management into their applications seamlessly. The endpoints are well-documented with Swagger annotations, making them easy to discover and use.

2.3.2.2 Smart Pricing

The Smart Pricing (SP) component belongs to WP4 and it is briefly included here for the sake of consistency. The reader is referred to D4.5 for a complete description of the component.

Description

The Smart Pricing (SP) component is an AI-driven system designed to optimize dynamic pricing decisions within the digital marketplace. By leveraging multi-agent reinforcement learning and game-theoretic auction mechanisms, the SP ensures competitive and balanced pricing that benefits both providers and consumers. Hosted securely on Eight Bells premises, the module operates autonomously, enabling trustworthy resource selection by determining fair market prices. Through continuous adaptation to market conditions, the SP fosters a competitive environment where providers adjust their prices dynamically, leading to optimized pricing outcomes.

Architecture

At its core, the SP functions as a multi-agent reinforcement learning system, where individual agents represent competing providers within the marketplace. These agents undergo training through a self-play mechanism, refining their pricing strategies over multiple iterations. The pricing process is governed by a multi-round blind reverse auction, inspired by game theory, where providers submit bids within predefined price boundaries. During each round, only their rank relative to competitors is disclosed, allowing them to adjust bids strategically. A key component within the SP is a load-balancing mechanism, which ensures a fair distribution of clients across providers, based on their quality of service and available resources. This mechanism favours less busy providers in the auction process, promoting balanced resource allocation and preventing network congestion. By integrating dynamic pricing, the SP reduces monopolistic pricing risks while maintaining an optimal balance between provider profitability and consumer affordability.

Interfaces

The SP shares final pricing decisions through a secure REST API that connects only to the Marketplace. This API acts as a trusted source of pricing data, which can be integrated into smart contracts or other digital transactions.

2.3.2.3 Oracles

Description

As previously introduced, the NANCY marketplace requires the interaction with non-Blockchain-based components. The way Blockchain can interact with these kinds of components is through *Oracles*.

An oracle is a component that provides a bridge between smart contracts deployed on a Blockchain network and external data sources. It essentially bridges off-chain and on-chain data. Since smart contracts cannot access data outside their Blockchain network, oracles are used to securely fetch and verify this external information. In NANCY, the marketplace requires interaction with two non-Blockchain-based components. These are the Smart Pricing (SP) and the Digital Agreement Creator (DAC), so two oracles are needed.

In addition, the marketplace also interacts with the SLA Registry smart contract. This interaction is usually directly managed inside the chaincode. However, there is a limitation in Fabric chaincode that avoids the generation of events if a chaincode is invoked from another chaincode. That means that the marketplace chaincode invoking the SLA Registry chaincode precludes the generation of an event from the SLA Registry. As this event generation is required in the inter-operator domain as explained in section 2.2.2.3, this limitation has been solved through the use of an intermediary oracle (SLA Signature oracle).

Architecture

The NANCY oracles have been implemented from scratch using the Fabric SDK⁸. Figure 12 shows the internal high-level architecture of the NANCY oracles.

⁸ [fabric-sdk package - github.com/hyperledger/fabric-sdk-go - Go Packages](https://github.com/hyperledger/fabric-sdk-go)

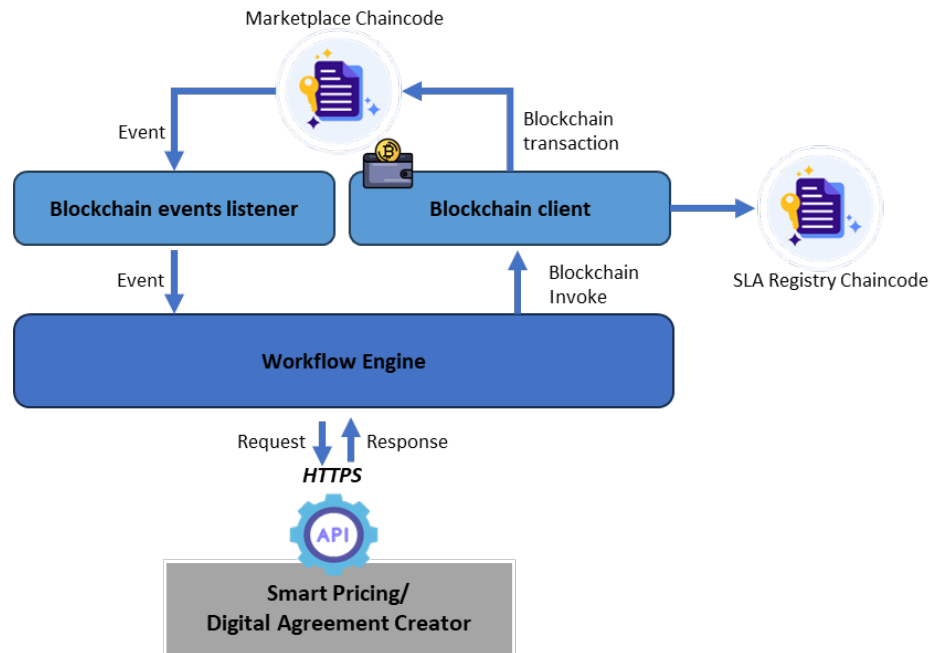


Figure 12. Blockchain oracles high-level architecture

Oracles have an internal Blockchain events listener since the marketplace interacts with them through Blockchain events. The oracle analyses the received event (*initPricing* and *initSLACreation* events), extracts the required information and notifies the SP and DAC components to start their operation inside its workflow engine. The interaction with these components is through HTTP requests to their exposed APIs, so the marketplace creates the required HTTP request to guarantee correct integration with the SP (API details are gathered in D4.5) and DAC (API details are gathered in section 2.3.2.1).

The SP and DAC components make their internal operation and provide the result as an HTTP response. This response is also analyzed by the workflow engine, which extracts the required data and processes it to feed the Blockchain client with the required information. The Blockchain client, which also contains a wallet, creates and signs the Blockchain transaction needed to interact with the marketplace and provides the result of the SP and DAC components operations. For this, oracles oversee updating the information in the marketplace chaincode regarding the SP and DAC responses through specific “Invokes” (see *setSearchPricing* and *setSearchSLA* in section 2.3.1.2).

In the case of the third oracle, which allows the correct interaction between the marketplace chaincode and the SLA Registry chaincode, it receives the event (*initSLASignature* event) through the Blockchain events listener, it extracts and processes the required information in the workflow engine to feed the Blockchain client where the required transaction needed to interact with the SLA Registry is created and signed (see *InitSLA* in section 2.3.1.1).

2.4 Other Features of the NANCY Blockchain

2.4.1 Blockchain Monitoring Dashboard

As described in Section 2.4.2.3, MITOSIS is a flexible approach to scale the blockchain by splitting a blockchain channel into multiple sub-channels and assign the nodes randomly to newly created channels. As more nodes join the system, the bigger size of the validator set will slow down the consensus process and in turn Impact the performance of the blockchain. Therefore, MITOSIS triggers “chain division” and creates two blockchains out of one by splitting the (large) set of validators in the original MITOSIS shard into two smaller subsets, so that the new blockchains have both a sufficiently small set of validators and can preserve low latency.

To further facilitate the chain-splitting process in MITOSIS, we extend the Hyperledger Fabric explorer to monitor the growth of the blockchain in real-time. By observing that one chain (i.e., Fabric channel) in the blockchain has too many peers running, the administrator can decide to split this channel and as well as to define how many organizations and peers should be assigned to each split channel.

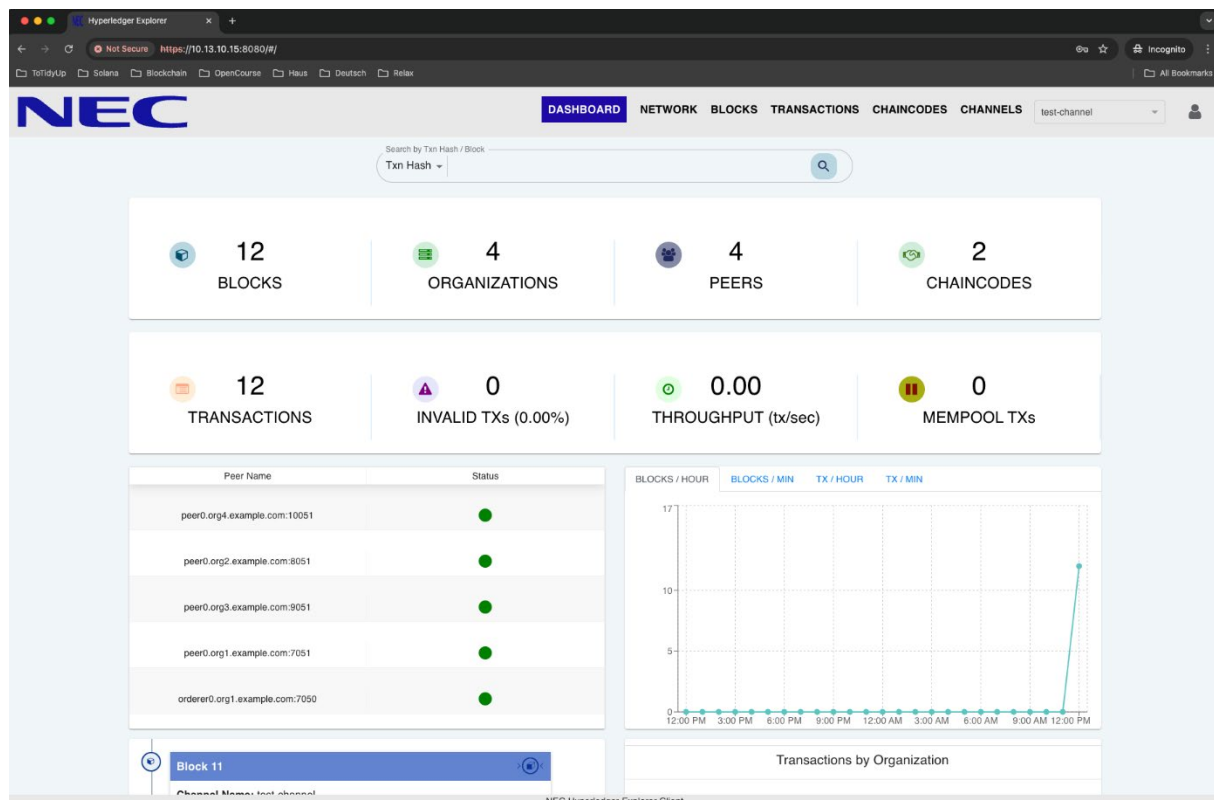


Figure 13 Fabric blockchain explorer

Figure 13 to Figure 15 show a demo of the blockchain monitoring dashboard. Figure 13 shows the blockchain explorer of the original channel ‘test-channel’ which is composed of 4 organizations and 4 peers. The control panel provides an option shown in Figure 14 to split the chain from the parent channel, in which the administrator can further define the size (i.e., the number of participants) of the new channels. Then it triggers a series of channel lifecycle transactions (see Figure 15) to modify the genesis block of the original channel to update the members in the channel, as well as create a new channel by assigning the members from the original channel.

Set Parameters for Chain Splitting


Name of the Parent Channel

Name of the Child Channel

List Of Organizations

Number of Peers per organization

Figure 14 Control panel of chain-splitting



DASHBOARD
NETWORK
BLOCKS
TRANSACTIONS
CHAINCODES
CHANNELS
test-channel

From To Search Reset

Select Orgs Select Installed Chaincodes Select Status

Creator	Created at	Channel Name	Tx Id	Type	Chaincode	Status
Org4MSP	02/05/2025, 12:18:04 PM	test-channel	8516fd...	ENDORSER_TRANSACTI...	_lifecycle	VALID
Org3MSP	02/05/2025, 12:18:04 PM	test-channel	2bac9f...	ENDORSER_TRANSACTI...	_lifecycle	VALID
Org2MSP	02/05/2025, 12:18:04 PM	test-channel	676485...	ENDORSER_TRANSACTI...	_lifecycle	VALID
Org1MSP	02/05/2025, 12:18:04 PM	test-channel	66bfa1...	ENDORSER_TRANSACTI...	_lifecycle	VALID
Org1MSP	02/05/2025, 12:18:04 PM	test-channel	1b629d...	ENDORSER_TRANSACTI...	chainManager	VALID
Org4MSP	02/05/2025, 12:18:04 PM	test-channel	3ab488...	ENDORSER_TRANSACTI...	_lifecycle	VALID
Org4MSP	02/05/2025, 12:18:04 PM	test-channel	42609c...	ENDORSER_TRANSACTI...	_lifecycle	VALID
Org3MSP	02/05/2025, 12:18:04 PM	test-channel	d9cb1a...	ENDORSER_TRANSACTI...	_lifecycle	VALID
Org2MSP	02/05/2025, 12:18:04 PM	test-channel	bc413a...	ENDORSER_TRANSACTI...	_lifecycle	VALID
Org1MSP	02/05/2025, 12:18:04 PM	test-channel	c31b56...	ENDORSER_TRANSACTI...	_lifecycle	VALID

Items per page 10 Page 2 of 3 < >

Figure 15 Lifecycle transactions to update channel information

2.4.2 Blockchain Scalability Mechanisms

Blockchain scalability refers to the ability of a blockchain network to handle an increasing volume of transactions without sacrificing performance. As blockchain technology continues to gain popularity, its ability to scale effectively is one of the most important challenges faced by developers and businesses.

Blockchain systems rely on a consensus mechanism operated by network nodes to establish a total order on transactions: by ensuring state changes are applied by all nodes in the same order, it guarantees that nodes have a consistent view of the blockchain state. In most blockchain systems, each transaction is processed by all network nodes and can be included in the ledger only if at least a

(super-)majority of the network nodes approve it. This makes blockchain systems secure but often slow when the number of users or transactions grows. As the demand for blockchain-based applications increases, the network must be able to process thousands of transactions per second (tps).

The main bottleneck in scaling permissionless blockchains such as Bitcoin and Ethereum is rooted in their relatively weak consistency property: A transaction is more likely to be stable the deeper it is in the ledger. In other words, although blocks are generated at a regular pace, blockchain participants cannot be certain that these blocks are stable in the ledger—they can only become more confident that a given block will not be reverted as more blocks are appended to it. Such a probabilistic guarantee implies a slow confirmation time—about 10 minutes for Bitcoin and 5 minutes for Ethereum—which severely limits their throughput (below 100 transactions per second (tx/s)).

Due to the slow confirmation times, permissionless blockchain systems suffer from latency and throughput limitations when compared to traditional state machine replication (SMR) systems (a.k.a. permissioned blockchains). For relying on classical consensus protocols, permissioned blockchains offer *finality*, meaning once a block is added to the blockchain it is permanent and cannot be rolled back. This feature makes permissioned blockchains a more appealing and faster alternative for practical use cases. Moreover, in a permissioned system, access and participation are restricted to authorized entities, which is ideal for enterprise use. As a result, leading industries and financial institutions are exploring permissioned blockchains to enhance their services and update their operations.

Classical consensus protocols, and hence permissioned blockchains, also face scalability challenges when it comes to the number of consensus nodes, which currently restricts their applicability to small- and medium-sized scenarios. Indeed, classical consensus protocols require multiple rounds of interaction among all participants before finalizing blocks in the ledger. Concretely, the communication complexity of state-of-the-art consensus protocols resilient to Byzantine faults is at least quadratic in the number of nodes [26].

2.4.2.1 Hardware-Assisted Byzantine-fault Tolerant Consensus

To improve the scalability of permissioned blockchains, different approaches have been explored. One such approach is to improve the complexity of the underlying consensus protocol by leveraging additional assumptions, e.g., network synchrony, availability of a public source of randomness, or trusted hardware. The NANCY blockchain uses fastBFT [27] as its consensus plugin, a BFT consensus protocol which reduces the communication rounds of PBFT [7] from 3 to 2 rounds and the communication message complexity from $O(n^2)$ to $O(n)$ by means of trusted hardware⁹ and cryptographic tools.

More specifically, the trusted secure hardware provides a trusted execution environment and confidentiality of sensitive data to desired applications, so that even a powerful adversary who takes over the whole system cannot read the private data or influence the execution process of the protected application. In this way, the critical part of the consensus protocol execution is protected using trusted hardware to limit the capabilities of a byzantine node. More concretely, in fastBFT, a malicious node cannot equivocate on the sequence number assigned to a produced message, i.e., announcing different order of the messages to different nodes. In this way, it reduces communication rounds during the agreement process. In addition, fastBFT improves the consensus voting process using lightweight cryptography: the nodes are organized in a tree structure and their votes are

⁹ In our concrete implementation, we use Intel SGX to fulfil the role of the trusted secure hardware.

aggregated using additional secret-sharing when votes are propagated to the leader node. Additional secret-sharing only involves XOR operation, which is much faster compared to aggregation based on digital signatures.

2.4.2.2 Benchmarking for NANCY Blockchain with fastBFT Consensus Plugin

We first integrate fastBFT to Hyperledger Fabric 2.2.4 and conduct the benchmarking using Hyperledger Caliper 0.4.2 [28]. Caliper is a benchmark framework for multiple blockchain platforms such as Hyperledger Fabric, Ethereum and FISCO BCOS. It is designed to benchmark transaction throughput and latency given using real-world application transactions. One can specifically define the smart contract to benchmark, and different transactions to the contract that involves multiple read or write operations. One can also define the number of clients that should be simulated and select the rate control strategy to issue client transactions.

In our benchmark setup, we build a channel composed of only one organization and one peer, so that the transaction proposal is only submitted to one peer before sending the proposal to the ordering service. We also use the following configuration (configtx.yaml) of the Fabric transaction batch as in Table 4, it is a trade-off between throughput and latency. For the benchmarking smart contract, we use the sample chaincode *fabcar*, which can be found in *fabric-samples* project release 2.2 [29].

Table 4. Fabric orderer configuration in configtx.yaml

Orderer Configuration	Value
BatchTimeout	3 seconds
BatchSize.MaxMesasgeCount	50
BatchSize.PreferredMaxBytes	512 KB

fabcar is a contract of an application to save and search data of different car models. For read operations, we call the transaction *queryAllCars*. Since querying transaction happens locally on a peer without submitting it to the consensus protocol, we only benchmark this read operation once with fix-load rate control, and this is around 1500 tx/s. Regarding write operations, we submit the transaction *createCar* with random inputs of approximately 100-200 Bytes. Moreover, we issue the transactions in the benchmark test with a fixed rate from 100 to 1200 tx/s (when applied) and test it on networks of 3, 5, 9, 15 and 21 orderers respectively. The benchmark test is conducted on one server with CPU Xeon E-2176G 3.7GHz with 12 vCore and 128 GB of RAM.

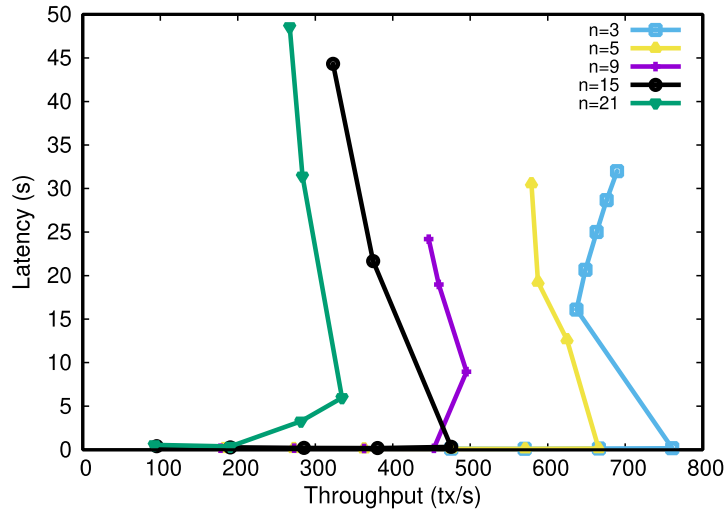


Figure 16 Transaction throughput and latency

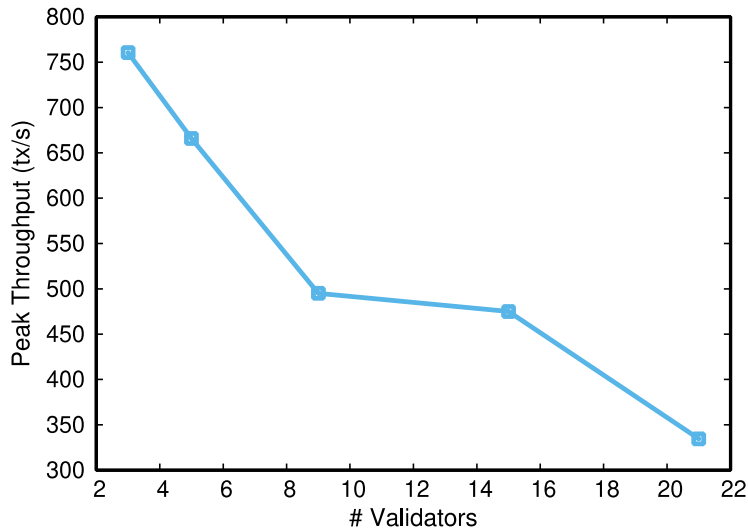


Figure 17 Peak throughput achieved by different network size.

We observe the measured transaction throughput and latency presented in Figure 16 and the peak throughput for each network size in Figure 17. With 3 orderers (validators), the write operation can reach a throughput of 766 tx/s and with 21 orderers. We found that during the benchmark test, the CPU is only up to 50% busy and RAM 20% occupied in all tests. This means that there can be some concurrent issues in the Caliper testing framework, which cause the resources to idle and not fully utilized.

2.4.2.3 Scalability Techniques for Permissioned Blockchains

Another promising technique to improve scalability is *blockchain sharding*, a general paradigm that involves using multiple blockchains in parallel, known as "shards," each running an independent instance of the same consensus protocol in a smaller network (with fewer nodes). The main idea is to operate parallel blockchain instances so that the transaction throughput increases by approximately a factor equal to the number of shards.

The NANCY blockchain leverages a variant of blockchain sharding, dubbed *MITOSIS*, specifically designed for dynamic environments in which the set of blockchain nodes may expand and shrink over time [21]. *MITOSIS* provides a methodology to create new blockchains recursively by splitting an existing blockchain into two child blockchains.

Similarly to sharding, *MITOSIS* leverages parallelism to enable higher participation (i.e., a higher number of blockchain nodes) while preserving the low latency of small-scale systems. Sharding requires all shards to run the same consensus protocol, and the numbers of shards and participants in each shard must be set at the protocol onset. In contrast to sharding, *MITOSIS* offers higher flexibility by allowing each blockchain in the ecosystem to select the consensus protocol of their choice, depending on specific requirements and trust assumptions, and it supports arbitrarily many shards and various shard sizes. By enabling the dynamic creation of heterogeneous shards, *MITOSIS* provides an attractive solution to scale permissioned blockchains in dynamic and fast-evolving environments.

MITOSIS facilitates the realization of a blockchain ecosystem with several blockchains running autonomously. Each such blockchain comprises a set of participants sharing a certain business logic, the participants comprising: clients, who issue transactions encoding specific service requests, and validators, who process client requests and include transactions in the blockchain ledger. The consensus protocol run by the validators in each *MITOSIS* shard is chosen by the participants of that shard independently of the other shards in the system, and each *MITOSIS* shard operates as an autonomous system. However, different *MITOSIS* shards can also interact with each other via cross-chain transactions, e.g., allowing users to exchange assets across shards. Interoperability among the various *MITOSIS* shards is enabled through dedicated functionalities to read from or write to blockchains of other shards.

As more nodes join the system, if the number of validators in a *MITOSIS* shard becomes significant, the latency of the underlying consensus protocol will decrease accordingly. In this case, *MITOSIS* triggers “chain division” and creates two blockchains out of one by splitting the (large) set of validators in the original *MITOSIS* shard into two smaller subsets, so that the new blockchains have both a sufficiently small set of validators and can preserve low latency.

The main challenge of the chain division process is to prevent faulty nodes from concentrating in one shard, as this may violate the trust requirements of the shard’s consensus protocol and hence fail to guarantee security. To prevent this scenario, *MITOSIS* assigns validators to the two created shards according to a randomized process.

MITOSIS Implementation

We implemented the idea of *MITOSIS* on top of Hyperledger Fabric. In what follows, we first give some background information about how multiple chains are supported and reconfigured in Fabric. Then we explain our approach to split a chain.

Fabric uses *channels* to support multiple independent chains. Each channel is configured by a consortium of organizations who first agree on the configurations of the chain out-of-band, e.g., the identities, certificates and anchor endpoints of each organization, the consensus protocol configurations and orderer endpoints, the chaincode endorsement policies, the access control policies, etc. The consented configuration is then encoded in a genesis block and is sent to the central registry service of Fabric to bootstrap the channel (chain). After the chain is created, chaincodes can then be

deployed to the chain and start accepting transactions. In Fabric, a chain can also be reconfigured once created, and any configuration update must be submitted to the central registry service as well.

Now we describe our approach to splitting a chain. We denominate the original chain as the *parent chain*, while the new chain which will be created is denominated as the *child chain*. Moreover, we define the size of a chain as the number of ordering service nodes participating in the consensus process. And we indicate with T the threshold size at which a chain split will be triggered. The threshold T must be decided by the organizations at chain creation time and depends on the consensus protocol used.

The general process of chain splitting includes first creating a copy of the ledger state of the parent chain and then performing a series of configuration updates that re-define the consortium of organizations in each chain. Then user accounts will be randomly assigned to either of the chains and therefore, their account state will be migrated to the newly assigned chain and will not have duplicates in both chains.

To first duplicate the ledger state of the parent chain, we use the `SNAPSHOT` feature [30] introduced by Hyperledger Fabric 2.3. `SNAPSHOT` enables the creation of an exportable representation of the state of a channel, which can then be used by a peer to join a channel without having to download the entire blockchain history. We use this feature to efficiently create a new channel at chain splitting time. More specifically, a snapshot contains:

- Public state of the network, which is in the form of (key,value) tuples which are stored in the world state database of the blockchain. It includes both application data and the configuration of the channel, however it does not contain historical values.
- Private data hashes and their configuration collections, which can be used by organizations to verify the private data that they receive. This is not relevant to our use of the snapshot.
- Past transaction IDs, which the peer uses to avoid accepting duplicated transactions that were previously broadcasted on the network.
- Metadata about the snapshot, which includes the hashes of the content of the snapshot, which can be used to verify its integrity, and the hash and number of the last block that was committed before the snapshot.

To achieve user account migration, a *ChainManager* chaincode must be first deployed to every chain that wish to be split in the future. *ChainManager* stores the configuration of all chains with which the “resident chain” (the chain in which the chainManager is instantiated) wants to communicate. In fact, the configurations of other chains are a critical piece of information when performing account migration, as it contains the information about the nodes and the system endorsement policies of other chains. With regard to how the configuration updates are sent to the chainManager, the protocol imposes no limitations; the configuration could be retrieved by communicating with honest peers, pulled from the centralized registry of configurations or a combination of both approaches. It is important to note that the configuration updates from other chains are always validated with the previous configuration. This step enables the chainManager to be sure that the new configuration he receives is legitimate. The genesis block of other chains instead, is subject to the voting process, since it cannot be verified with cryptography, and it is used as a root of trust for validating the subsequent configuration updates. The admins of all organizations need to express their approval of the relationship $\text{genesisBlock} \leftrightarrow \text{chainID}$. Once a vote has been expressed by all the organizations of the resident chain, the genesis block is considered trusted. Other than storing the configurations of other

chains, *chainManager* also provides the proof verification logic, which can be used to perform on-chain proof verification.

The split procedure involves the following steps:

1. *Schedule chain split process at a future block B.* When the number of ordering service nodes of a chain reaches the threshold T , we assume that at least one honest peer triggers the start of the split procedure. This assumption is reasonable since we assume that there is always an honest quorum of nodes in each chain. The split process is started by proposing a block number B which indicates the block at which the chain division will actually happen. An endorsement is then performed on this proposal, thus guaranteeing that all organizations agree on it. Once the proposal is endorsed by all organizations and thus the consensus on splitting at block B is reached, all peers autonomously schedule a snapshot to be performed at that block.
2. *Halt chain operations once block $B - 1$ is committed.* This means all transactions submitted after this point will not be endorsed by any peer. Note that block $B - 1$ will also be used in subsequent steps for the randomness computation.
3. *Construct block B which randomly assigns user accounts to either of the two split chains.* This operation is the only one which can be performed after the stop of the blockchain and before the actual split of the two chains happens. This is because the data structure representing a user's account contains a field which determines to which chain he belongs to. By changing this field we can change the channel on which the user is allowed to operate, since it is checked before performing each operation.
4. *Create the genesis block of the child chain based on the last configuration block of the parent chain.* More concretely, the last configuration block of the parent chain is fetched and the channel name in the configuration block is replaced with the name of the child chain.
5. *Prepare snapshot for the child chain.* Since snapshot is scheduled at block B , we wait for all peers to complete the snapshot generation process and then modify it to transport the state of the parent chain to the child chain. More concretely, the "last block hash" field in the metadata section is replaced with the hash of the previously fetched configuration block and the "last block number" is set to 0, since it is the start of a new chain.
6. *Agree on the hash of the snapshot.* At least one honest peer sends the hash of the snapshot to all organizations in the parent chain. The outcome of this step is a snapshot hash signed by all organizations, which can be used by newly joining peers to securely join the child chain. Moreover, both the genesis block of the child chain and the signed snapshot can be stored in the *chainManager* chaincodes of other chains, as well as in the central registry of configurations to increase availability.
7. *Bootstrap the child chain.* Organizations can now join the child chain by using the configuration block produced at step 4 as the genesis block. It is important to note that at this moment the world state of the child chain, which contains both application and configuration data, is exactly the same as the world state of the parent chain by using the snapshot. The parent chain has been successfully duplicated to the child chain.
8. *Randomly assign organizations to either chains.* We use the hash of block $B-1$ as the seed to perform the assignment randomly so that we can have a balanced and unbiased division of the organizations between the parent and the child chain. The pseudo-code of the algorithm can be referred to in Figure 18.
9. *Update configurations of both chains.* The configurations of both chains are now updated with the new assignment of organizations according to the previous step. Each configuration update is stored on the central registry and in the *chainManager* of both chains.

10. *Resume chain operations.* Now that the chain has been successfully split, both can be restarted and resume their operations.

```

randomness = Hash(block)
list_parentchain = []
list_childchain = []
parent_chain_full = child_chain_full = false
for orgID in sort(orgIDs):
    assignation = Hash(randomness | orgID)
    if assignation is odd:
        list_parentchain.append(orgID)
    else:
        list_childchain.append(orgID)
    if len(list_parentchain) == (total_number_of_orgs / 2):
        parent_chain_full = true
        break
    if len(list_childchain) == (total_number_of_orgs / 2):
        child_chain_full = true
        break
if parent_chain_full == true:
    child_chain.append(remaining_orgs)
if child_chain_full == true:
    parent_chain.append(remaining_orgs)

```

Figure 18 Pseudo-code of random assignment for organizations between parent chain and child chain

Note that to enable the above chain split procedure, some modifications to the code base of Hyperledger Fabric are necessary and the following components have been modified:

- **Configuration system chaincode.** We added a function to reach a consensus on the hash of a snapshot. It receives a channel ID and a block number, which together identify univocally the snapshot, as well as the hash of the snapshot. If the hash computed based on the local copy of the snapshot matches the input, the function returns true. Moreover, a function is added to compute at which block to perform the snapshot. The function takes a positive integer as input as the offset, and the sum of the current block height and the offset would be where the snapshot will be scheduled.
- **Lifecycle system chaincode.** We modify it to enable the commit of transactions coming from the csc system chaincode.

We further evaluate the time it takes to complete the chain-splitting process. Note that during the chain split, the liveness of the original blockchain suffers an interruption. Therefore, the performance result gives an indication of how long the blockchain service is not available to process any transactions. The resulting plot is provided in Figure 19. We see that the latency increases linearly with the size of the organization consortium. Since chain split is an occasional event, the interruption of several minutes is still acceptable.

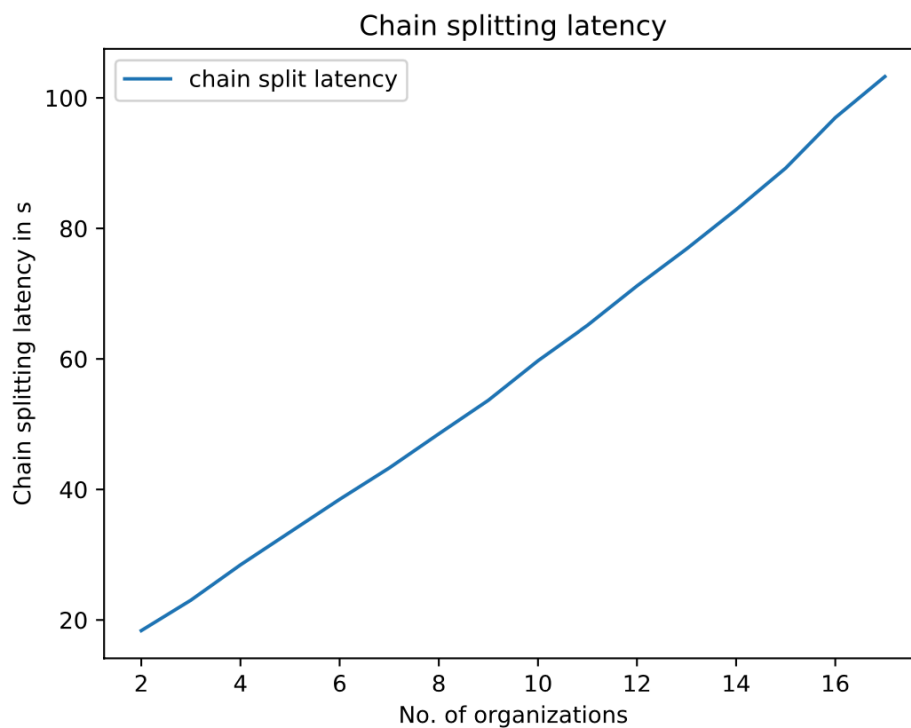


Figure 19 Latency of the chain split procedure versus number of organizations in the network.

2.4.3 Data Integrity Mechanisms

To provide data integrity for activities in the use cases, we use blockchain to timestamp any type of data for all partners to, later on, refer to the logged data. More specifically, we deploy a smart contract to simply record the id and the message digest of the data as a transaction without really saving them in the state. The message digest of the data must be yielded through a cryptographic hash function to avoid collision. The interface `CreateLog` only returns OK without recording the data in the state. When the submitted transaction is successfully validated, the partner must save the returned transaction Id for further reference. To later refer to the data, the partner can provide the transaction Id, which is timestamped in the blockchain and shows that the message digest of the data matches.

Chaincode Interface Description
<code>CreateLog (dataMD <i>string</i>)</code>
Returns OK.

2.4.4 Tokenization of Digital Assets

Tokenization of digital assets is a valuable sub-component for the NANCY blockchain to provide secure and flexible asset management. This sub-task focuses on enabling the creation, representation, and management of digital assets through tokenization mechanisms, adhering to blockchain standards for interoperability and transparency.

Description

The tokenization process provides a mechanism to digitally represent real-world or virtual assets as tokens on the blockchain. The key functionalities include:

1. **Smart Contract Development:**
 - Implementation of an ERC-721 compliant smart contract compatible with Hyperledger Fabric to enable non-fungible token (NFT) creation.
 - Definition of token attributes such as ownership, metadata, and uniqueness to represent digital assets accurately.
2. **Asset Storage Integration:**
 - Data is stored as assets in a decentralized file system (e.g., IPFS) while linking this data with the token on the blockchain.
 - Seamless access is ensured for tokenized assets through blockchain references.
3. **Asset-Token Association:**
 - Enable the binding of asset metadata to tokens using unique identifiers, ensuring traceability and authenticity.
 - Support for operations such as transfer of ownership, metadata updates, and token revocation.

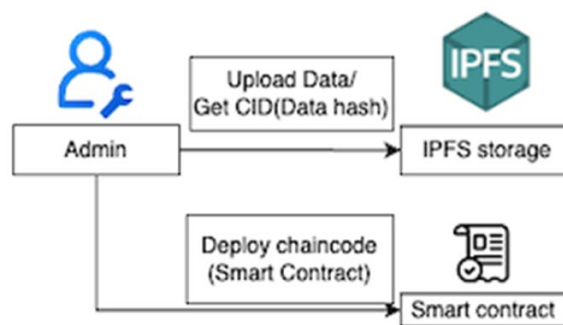


Figure 20. Admin prerequisites

Regarding the deployment of the contract, as shown in Figure 20, an Admin user can upload the digital assets to the IPFS and collect the corresponding CID (Content Identifier). Then, deploy the smart contract in the private network and associate the smart contract's minted (created) assets with the CIDs

Architecture

The tokenization system integrates smart contracts and a decentralized storage system to realize the tokenization module. More specifically, the following components:

- **Smart Contracts:** They reside on the blockchain network and handle token lifecycle management, including minting, transferring, and burning tokens.
- **Decentralized Storage:** Asset data can be securely stored in IPFS (or another decentralized storage solution), with references stored on the blockchain.


```

class ERC721Contract extends Contract
# Define Data Structures
struct NFT
  tokenId: String
  Owner: String
  TokenURI: String
  Approved: String

struct Approval
  Owner: String
  Operator: String
  Approved: Boolean

struct Transfer
  From: String
  To: String
  tokenId: String

# Contract Initialization
function Initialize(name: String, symbol: String) -> Error
  """
  Set the name and symbol of the contract.
  Ensure the contract is initialized only once.
  """
  if isAlreadyInitialized()
    return "Error: Contract is already initialized"
  storeContractMetadata(name, symbol)

# Mint a New NFT
function Mint(tokenId: String, tokenURI: String, owner: String) -> Error
  """
  Create a new NFT with a unique tokenId.
  Assign ownership to the specified owner.
  """
  if NFTExists(tokenId)
    return "Error: tokenId already exists"
  nft = NFT(tokenId: tokenId, Owner: owner, TokenURI: tokenURI, Approved: "")
  storeNFT(tokenId, nft)

# Transfer an NFT
function Transfer(from: String, to: String, tokenId: String) -> Error
  """
  Transfer ownership of the specified tokenId from one owner to another.
  Clear approval after the transfer.
  """
  nft = fetchNFT(tokenId)
  if nft == null or nft.Owner != from
    return "Error: Not authorized to transfer"
  nft.Owner = to
  nft.Approved = "" # Clear approval
  storeNFT(tokenId, nft)
  emit Transfer(From: from, To: to, tokenId: tokenId)

# Approve a Third Party for a Specific Token
function Approve(operator: String, tokenId: String) -> Error
  """
  Allow an operator to manage the specified tokenId on behalf of the owner.
  """
  nft = fetchNFT(tokenId)
  if nft == null
    return "Error: Token does not exist"
  nft.Approved = operator
  storeNFT(tokenId, nft)

# Check Operator Approval for All NFTs of an Owner
function IsApprovedForAll(owner: String, operator: String) -> Boolean
  """
  Verify if an operator is approved to manage all tokens of an owner.
  """
  approval = fetchApproval(owner, operator)
  return approval != null and approval.Approved

# Retrieve NFT Details
function GetNFT(tokenId: String) -> NFT or Error
  """
  Fetch details of an NFT using its tokenId.
  """
  nft = fetchNFT(tokenId)
  if nft == null
    return "Error: Token does not exist"
  return nft

# Retrieve Owner of an NFT
function GetOwner(tokenId: String) -> String or Error
  """
  Get the owner of a specific NFT using its tokenId.
  """
  nft = fetchNFT(tokenId)
  if nft == null
    return "Error: Token does not exist"
  return nft.Owner

# Retrieve Token Metadata
function GetMetadata(tokenId: String) -> String or Error
  """
  Fetch the metadata URI for a specific NFT.
  """
  nft = fetchNFT(tokenId)
  if nft == null
    return "Error: Token does not exist"
  return nft.TokenURI

# Check if an NFT Exists
function NFTExists(tokenId: String) -> Boolean
  """
  Determine if an NFT with the given tokenId exists.
  """
  return fetchNFT(tokenId) != null

# Burn an NFT
function Burn(tokenId: String) -> Error
  """
  Permanently remove an NFT from the ledger.
  """
  nft = fetchNFT(tokenId)
  if nft == null
    return "Error: Token does not exist"
  deleteNFT(tokenId)

# Retrieve All NFTs Owned by an Address
function GetBalance(owner: String) -> List of NFT
  """
  Fetch all NFTs owned by a specific address.
  """
  return fetchNFTsByOwner(owner)

# End of ERC721Contract

```

Figure 21. Smart contract pseudo-lang

Interfaces

- **Blockchain Smart Contracts:** These smart contracts effectively manage the token lifecycle and ensure compliance with the tokenization standards like ERC-721 and ERC-1155 token standards.
- **Storage Layer (IPFS):** Provides persistent storage for digital assets while maintaining decentralization and security.

The smart contract, with all the inherent functions in pseudo-language, is shown in Figure 21.

In Hyperledger Fabric, the primary distinction between invoke functions and query functions lies in their interaction with the blockchain ledger (see Table 5). Invoke functions are designed to modify the state of the ledger by performing operations such as creating, updating, or deleting records. These functions require endorsement and consensus from the network peers before the changes are committed to the blockchain, ensuring that all modifications are permanent and consistent across the network. On the other hand, query functions are read-only operations that retrieve data from the ledger without altering its state. They execute on a single peer and do not involve consensus, making them lightweight and efficient for data retrieval tasks such as checking ownership, balances, or metadata. While invoke functions are essential for state-changing operations like minting or transferring tokens, query functions are crucial for providing insights and visibility into the ledger's current state without impacting its data.

Table 5. Invoke vs Query Functions

Aspect	Invoke Functions	Query Functions
Effect on Ledger	Modify the ledger state	Do not change the ledger state
Transaction Process	Require endorsement and consensus	Do not require consensus
Persistence	Results are persisted to the blockchain	Results are transient (read-only)
Performance Impact	Involves full transaction lifecycle	Lightweight, executed on a single peer
Usage Examples	Minting tokens, transferring ownership	Checking balances, retrieving metadata

Let us present a more specific overview of the invoke and query functions displayed on the smart contract. The functions are categorized based on their types:

Invoke Functions

1. initialize(name: String, symbol: String)

- **Purpose:** Sets the name and symbol of the token collection and ensures the contract is initialized only once.
- **Usage:** Called during the deployment of the smart contract to define the collection's identity.

- **Key Checks:** (i) Prevents re-initialization. (ii) Only authorized entities can initialize the contract.
2. **mint(tokenId: String, tokenURI: String, owner: String)**
 - **Purpose:** Creates a new NFT with a unique identifier (tokenId) and assigns it to an owner.
 - **Usage:** Used to tokenize a new digital or physical asset.
 - **Key Actions:** (i) Ensures tokenId is unique and not already minted. (ii) Associates metadata (via tokenURI) with the token for descriptive purposes. (iii) Updates the owner's balance and token mappings.

Example: Tokenizing a digital artwork by assigning it a “tokenId” and linking it to a metadata URI.
 3. **transfer(from: String, to: String, tokenId: String)**
 - **Purpose:** Transfers ownership of an NFT from one user to another.
 - **Usage:** Called when the current owner wants to transfer or sell the NFT.
 - **Key Actions:** (i) Validates that the sender is either the owner or an approved operator. (ii) Updates ownership in the ledger. (iii) Clears any existing approvals for the token. (iv) Emits a Transfer event for traceability.
 4. **approve(operator: String, tokenId: String)**
 - **Purpose:** Grants approval to a third-party operator to manage a specific NFT.
 - **Usage:** Allows the operator to transfer the token on behalf of the owner.
 - **Key Actions:** (i) Verifies that the caller is the token owner. (ii) Updates the approval status in the ledger. (iii) Enables the operator to act on the owner's behalf for the specified token.
 5. **burn(tokenId: String)**
 - **Purpose:** Permanently removes an NFT from the ledger.
 - **Usage:** Used to retire tokens that are no longer needed or have fulfilled their purpose.
 - **Key Actions:** (i) Validates ownership. (ii) Deletes the token from the ledger. (iii) Emits a Transfer event indicating the token was burned.

Query Functions

6. **isApprovedForAll(owner: String, operator: String)**
 - **Purpose:** Checks whether an operator is authorized to manage all NFTs owned by a specific user.
 - **Usage:** Provides flexibility for users to delegate NFT management to trusted operators.
 - **Key Actions:** (i) Looks up the owner-operator approval mapping. (ii) Returns true if the operator has been granted global approval by the owner.
7. **getNFT(tokenId: String) -> NFT**
 - **Purpose:** Retrieves the details of a specific NFT, including its owner and metadata.
 - **Usage:** Used by clients or external systems to query the state of a token.
 - **Key Actions:** (i) Checks if the tokenId exists. (ii) Returns the NFT data if found.
8. **getOwner(tokenId: String) -> String**
 - **Purpose:** Returns the owner of a specific NFT.
 - **Usage:** Helps verify ownership during transactions or external queries.
 - **Key Actions:** (i) Fetches the owner information for the specified tokenId.
9. **getMetadata(tokenId: String) -> String**
 - **Purpose:** Retrieves the metadata URI associated with an NFT.
 - **Usage:** Provides descriptive details about the asset, such as its name, image, or other properties stored off-chain.

- **Key Actions:** (i) Ensures the token exists. (ii) Returns the TokenURI.
- 10. getBalance(owner: String) -> List of NFT**
- **Purpose:** Retrieves all NFTs owned by a specific user.
 - **Usage:** Enables users to query their portfolio of NFTs.
 - **Key Actions:** (i) Looks up all tokens associated with the given owner.
- 11. NFTExists(tokenId: String) -> Boolean**
- **Purpose:** Checks whether an NFT with the given tokenId exists.
 - **Usage:** Ensures that operations like transfers or updates are only performed on valid tokens.
 - **Key Actions:** (i) Queries the ledger for the token's existence.

Lifecycle events are notifications or logs emitted by a blockchain smart contract to indicate significant actions or state changes in the lifecycle of an asset or transaction. These events provide transparency, traceability, and integration capabilities by broadcasting important updates to off-chain systems or users. In the context of the ERC-721 Smart Contract, lifecycle events play a crucial role in managing and tracking the state of non-fungible tokens (NFTs). They ensure that stakeholders are informed about key actions, such as minting, transferring, or approving tokens, and enable external systems (e.g., applications, marketplaces) to react or synchronize with these changes.

Lifecycle Events

- 1. Mint Event (Transfer with from = 0x0):**
 - a. Triggered when a new NFT is created.
 - b. Indicates the initial assignment of ownership.
- 2. Transfer Event:**
 - a. Triggered when ownership of an NFT changes.
 - b. Provides details of the sender, recipient, and token.
- 3. Approval Event:**
 - a. Triggered when a third party is approved for a specific token or all tokens owned by a user.

The implementation of ERC-721 compliant smart contracts within the NANCY blockchain offers a robust and scalable solution for the tokenization of digital assets. By leveraging the unique capabilities of non-fungible tokens (NFTs), this approach guarantees secure ownership, traceability, and seamless interoperability of assets, whether digital or physical. To enhance privacy and monitoring capabilities, the solution is deployed on a private blockchain network, ensuring robust data protection while allowing comprehensive oversight of the network. This design effectively balances security, privacy, and functionality, enabling efficient asset management without compromising sensitive information.

2.4.5 Monitoring and Verification of the Transactions

Monitoring and verification are integral to the NANCY blockchain, ensuring transaction integrity, correctness, and transparency. *Verification* involves simulating transaction proposals on peers to validate compliance with endorsement policies and ensure ledger consistency, preventing invalid or unauthorized transactions from advancing. *Monitoring* complements this by tracking transaction activities, logging validation outcomes, reasons for rejections, and performance metrics, thereby enabling real-time visibility, auditing, and debugging.

In the NANCY blockchain, clients submit transaction proposals directly to peers for validation, bypassing the orderer and avoiding unnecessary computational overhead. Immediate feedback is provided to the client application, while a user-friendly Swagger UI and robust logging mechanisms enable users to query results, analyze metrics, and maintain full traceability. By integrating these mechanisms, the monitoring and verification processes ensure a secure, efficient, and accountable network, forming a critical foundation for the integrity and reliability of the NANCY blockchain system.

Description

The key functionalities of the proposed NANCY's monitoring and verification of transactions include:

- **Transaction Proposal Validation:** This enables client applications to create transaction proposals and send them to the appropriate peers for simulation and endorsement. Peers execute the transaction logic, validating (i) the endorsement policy compliance and (ii) read/write consistency with the current ledger state.
- **Result Feedback:** Collects the simulation results and endorsements from the peers. Moreover, it provides feedback to the client application on whether the transaction meets all validation criteria.
- **Transaction Metrics Logging:** Logs metrics for each validation attempt, such as the number of validated and rejected transactions, reasons for failure, and simulation results.
- **Efficient Monitoring:** Uses APIs exposed via a Swagger UI to enable users to interact with the system for querying transaction statuses, validation results, and other metrics.

Architecture

The architecture for the monitoring and verification system in the NANCY blockchain can be seen in Figure 22.

The system is designed to ensure secure, efficient, and transparent transaction validation by integrating key components at various layers. The **Client Application Layer** serves as the interface for generating transaction proposals and sending them to the blockchain peers for simulation. After validation, the client app receives feedback, including endorsements and validation results, to determine if the transaction can proceed to the orderer.

The **Blockchain Peers** handle the simulation and validation of transaction proposals. These peers execute the smart contract logic to validate compliance with endorsement policies and check ledger consistency, returning results to the client app without committing transactions to the ledger.

A **Metrics Logging** mechanism captures validation results and performance metrics, storing them in a centralized database for auditing, debugging, and performance analysis. Users can interact with the system through a **Swagger UI**, which exposes APIs for querying transaction statuses, validation outcomes, and detailed metrics.

- **Client Application Layer:** The client app generates transaction proposals and sends them to the appropriate peers for simulation. After validation, it receives feedback from the peers to determine if the transaction can be submitted to the orderer for ordering.
- **Blockchain Peers:** These peers simulate and validate the transaction proposal without committing it to the ledger. Then, the results of the simulation are returned, including endorsements and validation errors, to the client app.
- **Metrics Logging:** Validation results and metrics are logged in a centralized database for auditing and performance analysis.

Interfaces

- **Client Application:** The client app sends the transaction proposals to peers for validation. It also receives simulation results and endorsements to decide on further actions.
- **Blockchain Peers:** Executes smart contract logic and validates the transaction proposal without committing it.
- **Swagger UI:** This swagger provides APIs for querying transaction validation results and fetching metrics and simulation outcomes.

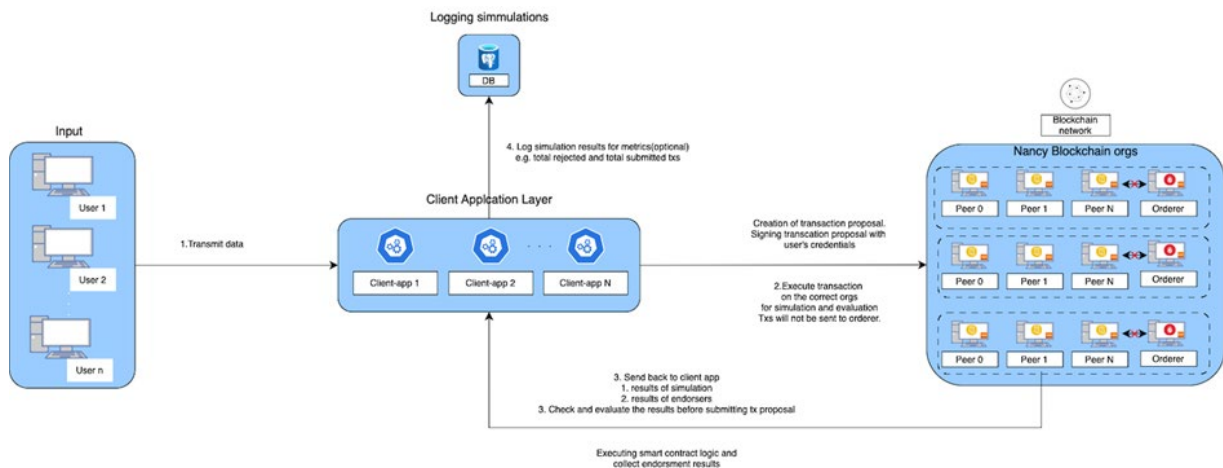


Figure 22. High-level architecture of the NANCY Transaction evaluation

By allowing transaction proposals to be validated without submission to the orderer, this task ensures the correctness and security of operations while reducing unnecessary computational overhead. The integration of a client application for transaction proposal evaluation, coupled with a Swagger UI for querying results, provides users with a streamlined and transparent interface for interaction. Furthermore, the logging and indexing mechanisms ensure complete traceability, enabling effective debugging, auditing, and performance analysis.

3 NANCY ID Management Tools

This section delves into the Self-Sovereign Identity and other privacy-oriented mechanisms researched in NANCY. It also describes the NANCY Wallet itself and its PQC Signature and SSI capabilities.

3.1 Introduction to SSI and wallets

Self-Sovereign Identity (SSI) is an approach to allow individual users to manage and control their own identities through a decentralized identity management system. In contrast to existing traditional identity management systems, where a centralized party (usually issuers), manage all the identities and credentials of a user and provides requested authentication services for the user to other application services (i.e., verifiers), SSI allows users to generate their own identities and keep the corresponding credentials locally, e.g., in a digital wallet, and handle authentication or authorization process directly with any application services. In this way, credential issuers will not be involved in every authentication process and thus improve efficiency as well as user privacy.

W3C has proposed corresponding standards for SSI systems, namely, the Decentralized Identifiers (DIDs) [31] and the Verifiable Credentials (VCs) [32]. DID provides a standardized approach to uniquely identifying users or subjects in decentralized systems, and VC describes a way to manage credentials, i.e., digitally signed attestations regarding a subject's attributes or affiliations, by leveraging DIDs for trust and interoperability. The standards propose an architecture where a user holds DIDs and VCs in his own digital wallet, requests issuers to acquire VCs, and interacts with verifiers to get authenticated by presenting Verifiable Presentations (VPs) derived from his VCs without disclosing his credentials. Meanwhile, all parties upload the public part of their identifiers and schemas in a verifiable data registry for other parties to lookup information (see the architecture in Figure 23).

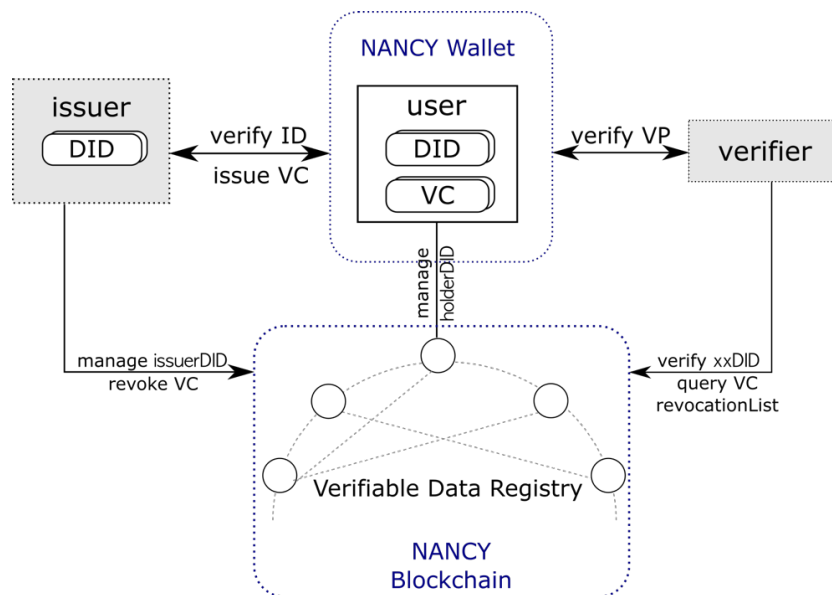


Figure 23. SSI Architecture with NANCY wallet and NANCY blockchain

The NANCY project functional requirements on privacy suggest the deployment of an SSI system for all providers and consumers. In this project, we use the NANCY blockchain as the verifiable data registry. All entities register their DIDs and the associated verification public keys in the blockchain. Issuers also publish the revoked or suspended VCs in the blockchain. We also provide every player with a wallet to manage their DIDs and VCs. Furthermore, the wallet also keeps the blockchain credentials used to interact with the NANCY blockchain, as it is a permissioned blockchain. For organizations such as

operators, the wallet also acts as a gateway that manages identities for multiple members from the same organization.

3.1.1 DID Registry and VC (Revocation) Registry

Description

As a verifiable data registry, the NANCY blockchain manages DID registration and VC revocation through two smart contracts. The DID registration is handled by smart contract *DIDRegistry*, which records DIDs and public keys associated with the DIDs. Meanwhile, another smart contract, *VCRegistry*, keeps a list of all revoked VCs for verifiers to look up during authentication.

Note that all smart contracts are implemented as Fabric chaincode in Golang. As introduced in Section 2.2, Fabric is a permissioned blockchain where all transaction issuers must first register by the CA of their organization to acquire an enrollment certificate (ECert) for contract invocations or queries. For both of our smart contracts, where access control policy is applied, we require that the identities registered in the ECert, i.e., *CommonName*, must align with the DIDs of the corresponding records that the transaction issuer wishes to access. More specifically, while DID is the form of `did:<did_method>:<id>`, the *CommonName* of the certification must be `<did_method>-<id>`.

Interfaces

In what follows, we present the interfaces of these smart contracts (see Table 6).

Table 6. Interface description of smart contract *DIDRegistry*

Chaincode Interface Description
DIDRegistry (didMethods []string)
is the constructor of the chaincode that defines a list of DID methods that are allowed to be recorded by this chaincode instance.
DID methods are the mechanism by which a particular type of DID and its associated DID document are created, resolved, updated, and deactivated. See more explanations in W3C standards.
DIDMethods ()
returns the list of DID methods that are allowed to be recorded by this chaincode instance.
register (did string, didDocument string)
adds a record of DID and its corresponding DID document (JSON string) in the registry. The chaincode must enforce access control decision that the transaction issuer possesses a valid ECert who is the controller of the corresponding DID. The input <code>didDocument</code> can be a boilerplate of the corresponding DID document.
resolve (did string)
returns the complete DID document of the queried DID <code>did</code> in JSON string.
update (did string, attrs ...string)

updates the record of `did` in the ledger with the supplied list of new attributes. The list of `attrs` are supplied in pairs (`attr, value`), in which `attr` is the path to an attribute in the DID document; and `value` is a JSON string that is used to overwrite the value of `attr`.

The chaincode must enforce access control decision that the transaction issuer possesses a valid ECert who is the controller of the corresponding DID. And `update` is not allowed to overwrite the "id" field of the DID document.

`delete(did string)`

removes a record of `did` from the ledger. The chaincode must enforce access control decision that the transaction issuer possesses a valid ECert who is the controller of the corresponding DID.

Now we define the *VCRegistry* chaincode to save the records of revoked or suspended Verifiable Credential (VCs).

Note that records of the registry are intended to be publicly accessible, we should take into consideration that the privacy of the VC content should not be compromised, even though they are revoked or suspended. For example, a revoked VC of a driver license still contains valid information about the name and birthday of a user; and we must leave this information out of the blockchain. Therefore, we only store a *reference* of the VC, for example, a cryptographic hash value of the VC, in the blockchain.

The constructor of the *VCRegistry* chaincode can define the criteria of the issuers, e.g., based on the attributes in their certificates, to apply finer access control rules on who is allowed to revoke VCs in this chaincode instance (see Table 7).

Table 7. Interface description of smart contract *VCRegistry*

Chaincode Interface Description
<code>revoke(vcRef string, issuerDID string)</code>
records that a VC with <code>vcRef</code> is revoked by the issuer <code>issuerDID</code> . The chaincode must enforce access control decision that the transaction issuer possesses a valid ECert that is issued to <code>issuerDID</code> . Then the revoked <code>vcRef</code> is recorded in the ledger, along with the <code>issuerDID</code> . Note that <code>issuerDID</code> must be stored along, as the chaincode cannot verify if the input <code>issuerDID</code> is indeed the issuer of the corresponding VC of the input <code>vcRef</code> . It is the responsibility of the querier to validate this information when retrieving the revocation list.
<code>suspend(vcRef string, issuerDID string)</code>
records that a VC with <code>vcRef</code> is suspended by issuer <code>issuerDID</code> in a similar way as <code>revoke</code> .
<code>lookupVCStatus(vcRef string, issuerDID string)</code>
returns the status of VC, whether it is <i>revoked</i> , <i>suspended</i> or <i>NA</i> . <i>NA</i> means the queried VC is neither revoked or suspended as recorded in the registry.
<code>getRevocationList(issuerDID string)</code>
returns a list of <code>vcRef</code> that are revoked by <code>issuerDID</code> . The input <code>issuerDID</code> is optional and if empty, return the full revocation list.


```
getSuspensionList(issuerDID string)
```

returns a list of `vcRef` that are suspended by `issuerDID`. In a similar way as `getRevocationList`.

3.2 The NANCY Wallet

The NANCY Wallet `WALLETGATEWAY` creates and holds the DIDs and credentials for the users, i.e., providers and consumers. In addition, the wallet, as integrated with each relevant component (e.g. UEs, SOs, BSS, others), runs a gRPC service to communicate with the blockchain. As explained in 2.3.1 and 3.1.1, the NANCY wallet gateway has defined specific gRPC methods to interact with the marketplace and the DID registry smart contracts.

Users can interact with the wallet gateway service with any gRPC implementation, for example, `grpcurl`. To improve the usability, we also provide a wallet client implementation `WALLETCLIENT` so that users can talk to the wallet with command line options. The usage of `WALLETCLIENT` is defined in Section 3.2.3.4.

The NANCY wallet has two roles: (1) *User Equipment (UE)* wallet or (2) *non-UE* wallet. Note that consumer users can be UE or non-UE, depending on whether the consumer is an operator, but providers are all non-UEs.

Each wallet is initialized with a `uid` provided by the user, where the `uid` has to be locally unique in each wallet storage. The `uid` is used as an alias for the user to restart the wallet later. When a wallet is initialized with a `uid`, it automatically creates a DID in the form of `did:nancy:<uid>-<pubkey_md>`. `pubkey_md` is the message digest (the first 16B of the Base58 encoding of SHA256) of the public key of the DID key pair, where the DID key pair for verification is either randomly generated based on ECDSA256 for a non-UE wallet, or retrieved from the PQC hardware token by querying through the specified APIs defined in Section 3.2.3.1 if it is a UE wallet. Note that each UE wallet holds only one PQC key pair and thus all associated DIDs will refer to the same PQC key material. The message digest of the public key is salted with a random value before being hashed, so that the combined DID string is anonymous as well as globally unique. Since Fabric certificates (ECert) refrain from the usage of some special symbols including “:”, the full DID string cannot be used as the identity of a Fabric certificate, therefore, we escape the prefix and only use `nancy-<uid>-<pubkey_md>` as the enrollment id of the ECert.

As a gateway to the blockchain, the wallet serves as a registrar of the CA, and it registers as well as enrolls each user to the blockchain. The enrollment certificates that the wallet acquires from the blockchain are saved to the local `/wallet/` directory.

3.2.1 PQC Signature Capabilities of the NANCY Wallet

A PQC Digital Signature Solution was developed, composed of:

- **PQC Signature Token:** it consists of a smart card integrating a quantum-resistant digital signature algorithm. TDIS follows the ongoing initiative from NIST to standardise a set of quantum-resistant algorithms¹⁰. This feature can be used to ensure the integrity and authentication of NANCY blockchains. For this purpose, asymmetric key pairs are used: the

¹⁰ [Post-Quantum Cryptography | CSRC \(nist.gov\)](https://www.nist.gov/post-quantum-cryptography)

private keys are stored in the token and used to sign. The signature is internally processed and based on the digest of the block data. On the other hand, the associated public keys allow the receiver to check the signature and, thus to verify the authenticity and integrity of the block.

- **PQC Signature Middleware:** this middleware (or driver) provides minimal services to the NANCY wallet for interfacing with the token.
- **JAVA SDK:** this toolkit provides the necessary wrappers as the NANCY wallet is developed in Java language.

The PQC Digital Signature Solution presents the following features:

- Frugal implementation of PQC Digital Signature on tiny CPU devices environment (32bits CPU, 24kB RAM)
- Selected PQC algorithm: Crystals Dilithium-SHAKE targeting security level level 3 as recommended by NIST agency
- High secure implementation of the cryptographic algorithm including countermeasures against state-of-the-art attacks (side channel, fault injection attacks)
- Acceptable performance compared to classical cryptography
- Hybrid concept that consists of a combination of pre-quantum and post-quantum cryptographic algorithms

Extended information can be found in NANCY D5.1.

3.2.2 SSI Capabilities of the NANCY Wallet

For each user account created in Wallet, an anonymous DID is automatically created for that user in form `did:nancy:<uid>-<pubkey_md>` and registered in the blockchain via DIDRegistry. Moreover, the wallet provides standard interfaces for manual management of DIDs and VCs. More details can be found in Section 3.2.3.2 and the interface description in Section GRPC API for SSI.

3.2.3 Architecture and Interfaces

This section includes descriptions of the architecture and interfaces for the:

- PQC capabilities of the NANCY wallet
- SSI capabilities of the NANCY wallet

3.2.3.1 Architecture overview for the PQC capabilities of the NANCY wallet

As shown in Figure 24, the solution is composed of two basic elements:

- PQC Signature Token: smart card implementing the PQC digital signature
- PQC Signature Middleware: contains the Driver offering easier access to the token from the upper Applications

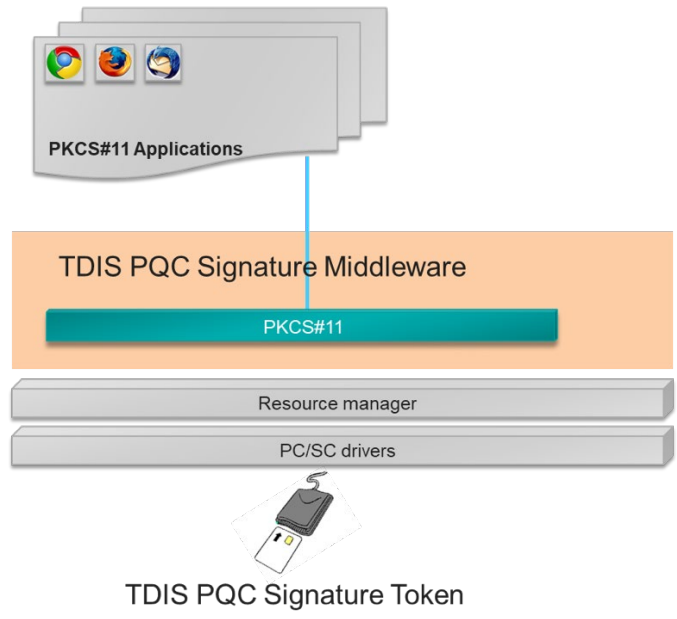


Figure 24: Architecture of the PQC Signature Solution

PQC Signature Middleware

The PQC Signature Middleware published to the Application layer a set of APIs commonly used in PKI systems namely PKCS#11¹¹ and used for digital signature purposes. The provided PQC Signature Middleware is running on a Linux PC. Resource Manager & PC/SC drivers are the market low-layer drivers to facilitate access to the smart cards. The communication interface with the smart cards can be Contact (ISO/IEC 7816) or Contactless RFID (ISO/IEC 14448).

PQC Signature Token

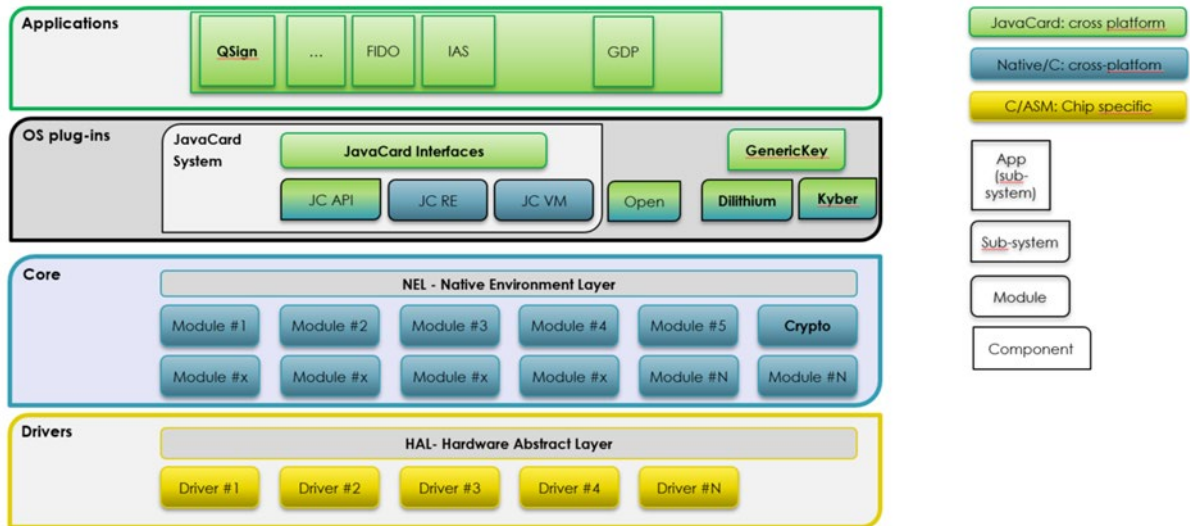


Figure 25: Architecture of the PQC Signature Token

Figure 25 represents the architecture of the PQC signature device. At the top of the diagram, we see the *application layers*, including the target electronic signature application, named QSign (Quantum

¹¹ [Workspace Home - OASIS \(oasis-open.org\)](https://www.oasis-open.org/Workspace/Home)

Signature) managing all the PKI operations. This application will rely on JavaCard APIs (standard and proprietary), implemented by the upper layers of the Operating System.

At the lower layers, we find the *cryptographic primitives* Dilithium and Kyber. Finally, we find the Hardware Chip containing the CPU, RAM and NVM memories, Cryptographic hardware accelerations, security sensors, etc. The communication interface with the external system can be contacted (ISO/IEC 7816) or contactless RFID (ISO/IEC 14448).

As shown in Figure 26, a Java SDK provides the necessary Java to Native wrappers since the NANCY wallet is developed in Java language.

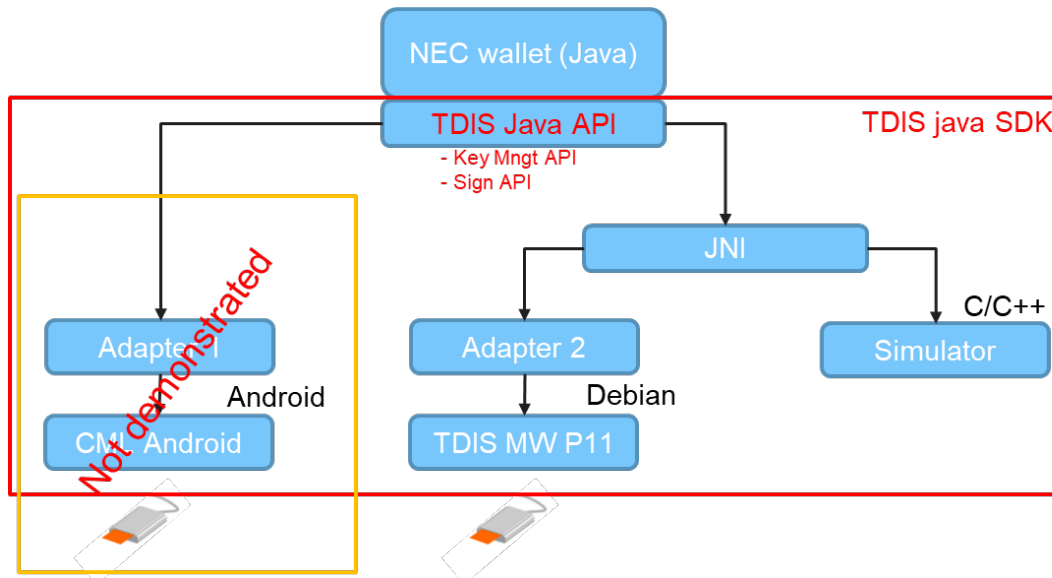


Figure 26: NANCY PQC Java SDK

PQC Signature Middleware Communication Interfaces

The PQC Signature Middleware publishes a set of APIs allowing the upper application to:

- Generate a new key pair
- Create a Signature

Figure 27 represents the sequence diagrams for the communication during a *key pair generation*. This sequence is not used in NANCY Demonstrators as PQC Signature Tokens are personalized at TDIS factory, but it is shown here since the project has researched it for a real deployment.

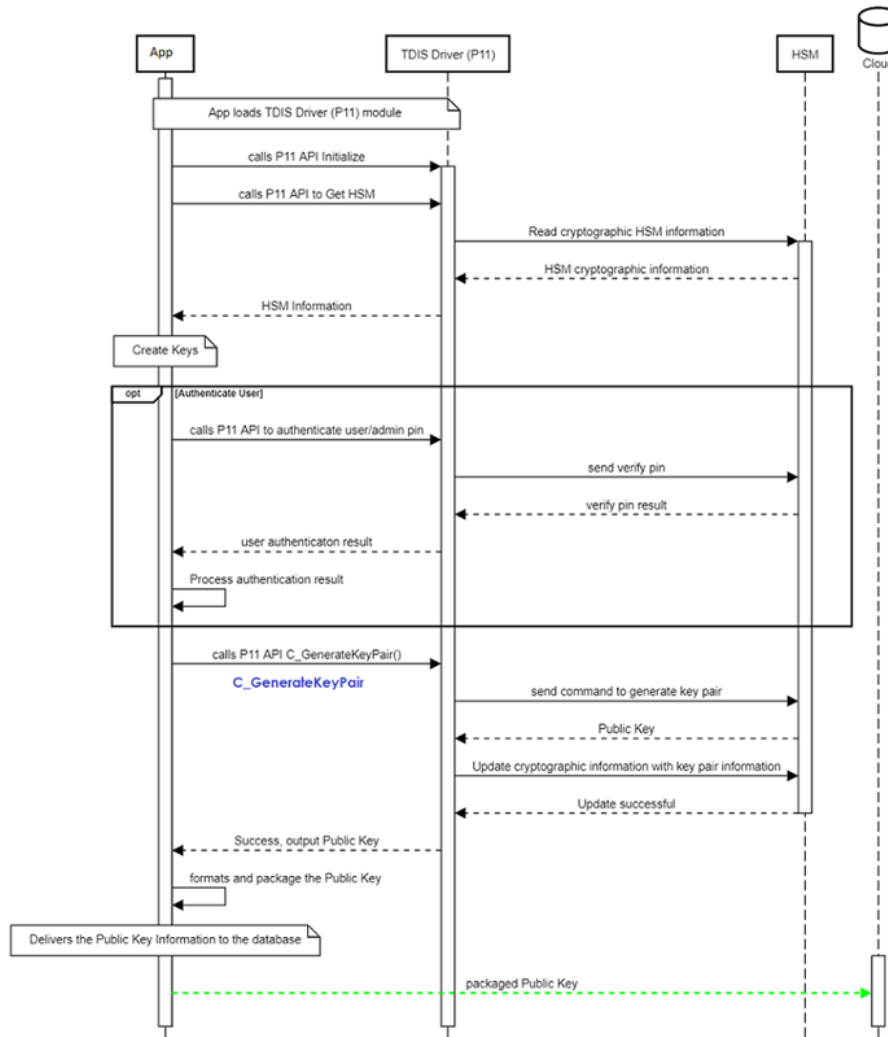


Figure 27: Generating a key pair

Figure 28 represents the sequence diagram for the *PQC signature* capability.

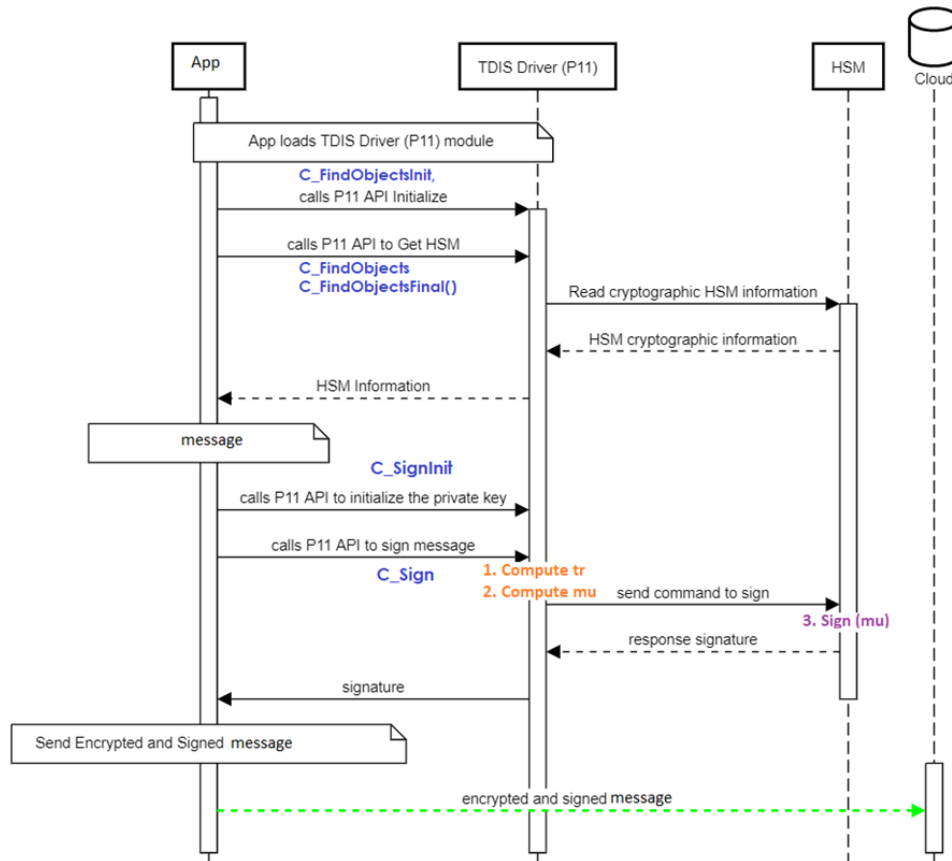


Figure 28: Signature Creation

3.2.3.2 Architecture overview for other capabilities of the NANCY wallet (SSI and interaction with the Marketplace, including SLAs)

Figure 29 illustrates the interaction between the wallet client, the wallet gateway, and the smart contracts in the NANCY blockchain. `WALLETGATEWAY` maintains all the credentials for a user and runs a gRPC service that can issue transactions to or query data from the blockchain. Users can access the wallet via any gRPC clients such as `grpcurl` and Postman. We provide a `WALLETCLIENT` implementation that simplifies the queries.

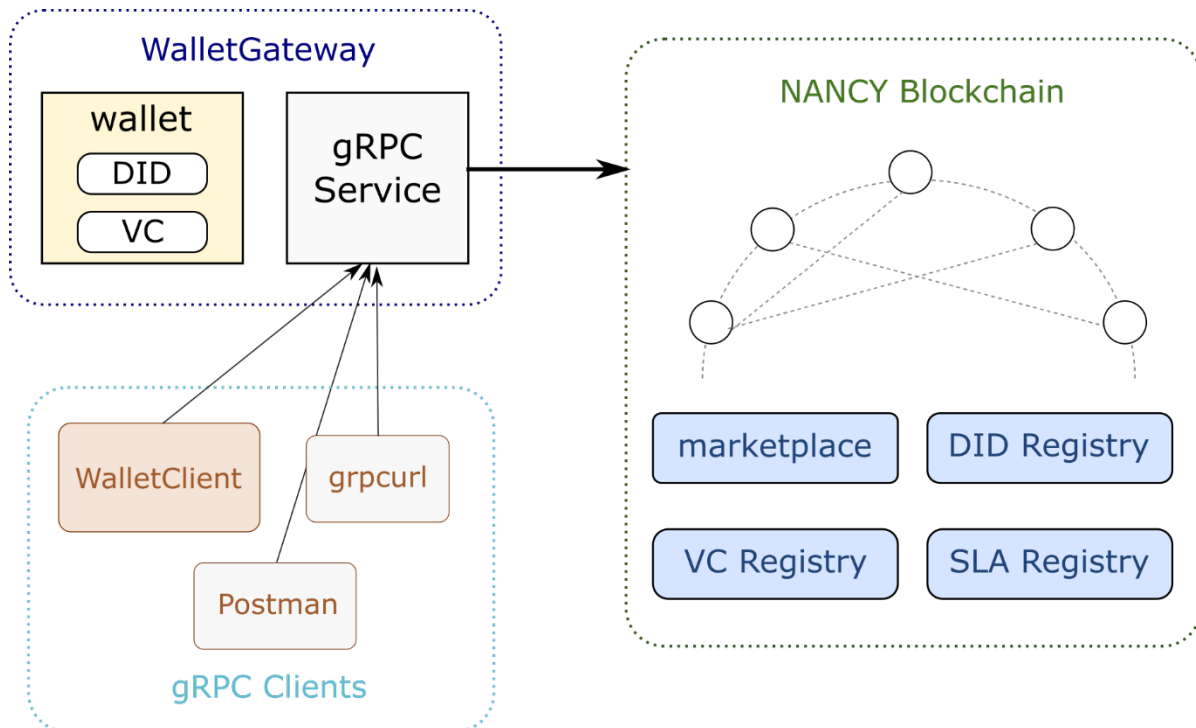


Figure 29 Architecture of the interaction between the wallet and the NANCY blockchain

WALLETGATEWAY Interfaces

In what follows, we first present the gRPC methods of the **WALLETGATEWAY** service, which can be queried using any gRPC implementation such as **grpcurl**. Then we present the usage of the command line tool **WalletClient** that can talk with the **WALLETGATEWAY** service through the gRPC methods described.

3.2.3.3 gRPC Service of WALLETGATEWAY

The wallet gateway service is called **dlt.DltGatewayService**. It has defined the following gRPC methods:

gRPC API for SSI

Issuer Methods

CreateCredential

Creates a verifiable credential with the provided `properties` for the holder with the given `holderDID`.

Request: `dlt.CredentialParam`

- ``string holderDID`` - The holder's DID.
- ``string properties`` - The properties in JSON format.

Example of properties in JSON format:

```
{  
  "name": "Bob",  
  "surname": "Smith",  
  "age": 24,  
  "gender": "non-binary"  
}
```

Response: *dlt.Credential*

- ``string` vcRef` - The credential reference.
- ``string` credential` - The credential in JSON format.
- ``string` error` - The error message if any.

Example of credential in JSON format:

```
{  
  "@context": [  
    "https://www.w3.org/2018/credentials/v1",  
    "https://www.w3.org/2018/credentials/examples/v1"  
  ],  
  "type": [  
    "VerifiableCredential",  
    "NancyCredential"  
  ],  
  "id": "VerifiableCredential-1725877365488",  
  "issuer": "did:nancy:issuer-A",  
  "issuanceDate": "2024-09-09T10:22:45Z",  
  "expirationDate": "2029-06-17T18:56:59Z",  
  "credentialSubject": {  
    "id": "did:nancy:holder-1",  
    "userType": "userEquipment",  
    "priority": "high",  
    "budget": "1000"  
  },  
  "proof": {
```



```

"type": "EcdsaSecp256k1Signature2019",
"created": "2024-09-09T10:22:45Z",
"domain": "example.com",
"nonce": "T1m8G52e5oiNjLA",
"proofPurpose": "assertionMethod",
"verificationMethod": "did:nancy:issuer-A#key-1",
"jws":
"eyJJiNjQiOmZhbHNILCJjcmI0IjpbImI2NCJdLCJhbGciOiJFUzI1NksifQ..1EcFzoO95atggYsKZje5I3DLaphWhBwKP
Miuvq2_XwlaDNh8UChkx7yx6UBMSX4r7IY68yAZleFUSliHdhyx7g"
}
}

```

ListCredentials

Returns a serialized JSON list of the credentials created by the queried issuer.

Request: *google.protobuf*.

An empty request parameter

Response: *dlt.Response*

- *string* value - List of credentials in JSON format.

- *string* error - The error message if any.

Example of the list of credentials in JSON format.

```

[
{
  "holderDID": "did:nancy:holder-1",
  "properties": "{\"userType\":\"userEquipment\",\"priority\":\"high\",\"budget\":\"1000\"}",
  "vcRef": "6dc687a9dd37119497029fb018bcb160a3bba7e3fc59629667fa80752b5403d7",
  "revoked": true
},
{
  "holderDID": "did:nancy:holder-2",
  "properties": "{\"userType\":\"userEquipment\",\"priority\":\"low\",\"budget\":\"4000\"}",
  "vcRef": "5556dc6ddegrg54dd37119497029fb018bcb160a3bba7e3fc5962wefewfgteh",
  "revoked": false
}
]

```

RevokeCredential

Revokes a credential that was issued by the queried issuer.

Request: *dlt.Request*

- *string* value - The credential reference (the vcRef received returned in `CreateCredential` method).

Response: *dlt.Response*

- *string* value - Empty.

- *string* error - The error message if any.

- Holder Methods

CreatePresentation

Creates a verifiable presentation. It is necessary to get the nonce from the verifier before.

Request: *dlt.PresentationParam*

- *string* credential - The credential in JSON format.

- *string* nonce - The nonce.

Response: *dlt.Response*

- *string* value - The presentation in JSON format.

- *string* error - The error message if any.

Example of presentation in JSON format:

```
{
  "@context": [
    "https://www.w3.org/2018/credentials/v1"
  ],
  "type": [
    "VerifiablePresentation"
  ],
  "verifiableCredential": {
    "@context": [
      "https://www.w3.org/2018/credentials/v1",
      "https://www.w3.org/2018/credentials/examples/v1"
    ],
    "type": [
      "VerifiableCredential",
```

```

    "NancyCredential"
  ],
  "id": "VerifiableCredential-1725877365488",
  "issuer": "did:nancy:issuer-A",
  "issuanceDate": "2024-09-09T10:22:45Z",
  "expirationDate": "2029-06-17T18:56:59Z",
  "credentialSubject": {
    "id": "did:nancy:holder-1",
    "userType": "userEquipment",
    "priority": "high",
    "budget": "1000"
  },
  "proof": {
    "type": "EcdsaSecp256k1Signature2019",
    "created": "2024-09-09T10:22:45Z",
    "domain": "example.com",
    "nonce": "T1m8G52e5oiNjLA",
    "proofPurpose": "assertionMethod",
    "verificationMethod": "did:nancy:issuer-A#key-1",
    "jws":
"eyJiNjQiOmZhbHNILCJjcmI0ljbImI2NCJdLCJhbGciOiJFUz11NksifQ..1EcFzoO95atggYsKZje5I3DLaphWhBwKP
Miuvq2_XwlaDNh8UChkx7yx6UBMSX4r7IY68yAZleFUSliHdhyx7g"
  }
},
  "proof": {
    "type": "EcdsaSecp256k1Signature2019",
    "created": "2024-09-09T10:23:30Z",
    "domain": "example.com",
    "nonce": "IerPI5rCX7Ap",
    "proofPurpose": "assertionMethod",
    "verificationMethod": "did:nancy:holder-1#key-1",
    "jws":
"eyJiNjQiOmZhbHNILCJjcmI0ljbImI2NCJdLCJhbGciOiJFUz11NksifQ..x0kUqk66E2aAcIjVXO8vZ1hnmqAWfOVr2
FYb0T0z9iAdke1c_8e8qlxVMsUkSHPfUF0FWBJmTwWPK3-9-JkaEQ"
  }
}

```

```

}
}

```

○ Verifier Methods

VerifyCredential

Verifies a credential. It is necessary that `GetNonce` is first called for the holder associated with the requested VP.

Request: *dlt.Request*

- ``string value`` - The verifiable presentation in JSON format.

Response: *dlt.Verification*

- ``string error`` - The error message (e.g. Network error, Blockchain error)

- ``string cause`` - The reason for failure (e.g. the credential was revoked, the signature is wrong).

- ``bool result`` - The result of the verification.

GetNonce

The verifier creates a nonce associated with the queried holderDID so it will be used for verification of the VP later on.

Request: *dlt.Request*

- ``string value`` - The DID.

Response: *dlt.Response*

- ``string value`` - The nonce associated to the queried DID.

- ``string error`` - The error message if any.

Following are the GRPC methods for interacting with the Marketplace (providers and services).

GRPC API for Marketplace

○ Methods to manage providers:

The data model for providers is:

- `name`: Name of the provider. Type: string. Ex: "vodafone"
- `type`: Type of providers. Type: string. Enum with values: ["operator", "publisher"]

The search component keeps track of the status of the request of a user for Service.

CreateProvider

Registers a new provider.

Request: *dlt.Request*

- ``string value`` - The provider details in JSON string.

Example of JSON:

```
{
  "id": " did:nancy:123-CwDuTckhy2D5VicHSFbVQG ",
  "name": "orange",
  "type": "publisher"
}
```

Response: *dlt.Response*

- ``string value`` - The provider in JSON string.
- ``string error`` - The error message if any.

Example of response "value" in JSON format:

```
{
  "id": " did:nancy:123-CwDuTckhy2D5VicHSFbVQG ",
  "model_type": "nancy_provider",
  "model_version": "0.1.2",
  "owner": "54f23c05b8c66125c870cd0c89bf7c08978fcc28c0cce126be78c5ad0887ef78",
  "mspi_id": "org0-nancy-dev",
  "timestamp": 1725538763,
  "name": "orange",
  "type": "publisher"
}
```

GetProvider

Returns the provider details given the provider ID.

Request: *dlt.Request*

- ``string value`` - The provider id.

Response: *dlt.Response*

- ``string value`` - The provider in JSON string.
- ``string error`` - The error message if any.

Example of provider in JSON format:

```
{
  "id": "did:nancy:123-CwDuTckhy2D5VicHSFbVQG",
  "model_type": "nancy_provider",
  "model_version": "0.1.2",
  "owner": "54f23c05b8c66125c870cd0c89bf7c08978fcc28c0cce126be78c5ad0887ef78",
}
```

```

"mspi_id": "org0-nancy-dev",
"timestamp": 1725538763,
"name": "orange",
"type": "publisher"
}

```

UpdateProvider

Updates the attributes of a provider.

Request: *dlt.Provider*

- ``string providerId`` - The provider id.
- ``string json`` - A request with the following format. Type JSON.

```

{
  "type": "operator2"
}

```

Response: *dlt.Response*

- ``string value`` - The provider details in JSON format.
- ``string error`` - The error message if any.

ListProvider

Lists all providers that match the filter.

Request: *dlt.Request*

- ``string value`` - The filter to apply for retrieving the list of providers.

Example of filter JSON:

```

{
  "id": { "$regex": ".*" }
}

```

Response: *dlt.Response*

- ``string value`` - The list of filtered providers in JSON format.
- ``string error`` - The error message if any.

Example of list of providers:

```

[
  {
    "id": "did:nancy:123-CwDuTckhy2D5VicHSFbVQG",
    "model_type": "nancy_provider",

```

```

    "model_version": "0.1.0",
    "owner":
    "eDUwOTo6Q049dXNlcjEub3JnMC5uYW5jeS5kZXYsT1U9Y2xpZW50K09VPW9yZzErT1U9ZGVwYXJ0bWVudD
    E6OkNOPWNhMS5vcmcwLm5hbmN5LmRldixPPW9yZzAubmFuY3kuZGV2LEw9UmFsZWlnaCXTVD1Ob3J0aC
    BDYXJvbGluYSxDPVVT",
    "mspi_id": "",
    "timestamp": 1721771340,
    "name": "vodafone",
    "type": "operator"
  },
  {
    "id": " did:nancy:456-GzerZ4WEgEM2Avx9zmMRUr",
    "model_type": "nancy_provider",
    "model_version": "0.1.0",
    "owner": "54f23c05b8c66125c870cd0c89bf7c08978fcc28c0cce126be78c5ad0887ef78",
    "mspi_id": "org0-nancy-dev",
    "timestamp": 1721773759,
    "name": "vodafone",
    "type": "operator2"
  }
]

```

DeleteProvider

Deletes a provider given the provider ID.

Request: *dlt.Request*

- `string value` - The provider id.

Response: *dlt.Response*

- `string value` - The provider id.

- `string error` - The error message if any.

○ Methods to manage services

The data model for services is:

- *providerID*: The ID of the provider the service belongs to. Type string
- *minPrice*: Minimum price. Type float64
- *maxPrice*: Maximum price. Type float64
- *Duration*: Duration. Type string

- *ResponseTime*: Response time. Type string
- *Throughput*: Throughput. Type string
- *Latency*: Latency. Type string

CreateService

Creates a service under a given provider.

Request: *dlt.Request*

- *string* value - The service request in JSON format.

Example of service in JSON format:

```
{
  "provider_id": "did:nancy:123-CwDuTckhy2D5VicHSFbVQG",
  "minPrice": 40.00,
  "maxPrice": 120.00,
  "duration": "60",
  "responseTime": "30",
  "throughput": "0.75",
  "latency": "32"
}
```

Response: *dlt.Response*

- *string* value - The service in JSON format.

- *string* error - The error message if any.

Example of service in JSON format:

```
{
  "id": "36e6049e9a8546a65344da94969ee9123f9df9e01d7b8ffab0eb4e30e0625a47",
  "model_type": "nancy_service",
  "model_version": "0.1.3",
  "owner": "54f23c05b8c66125c870cd0c89bf7c08978fcc28c0cce126be78c5ad0887ef78",
  "mspi_id": "org0-nancy-dev",
  "timestamp": 1725538790,
  "provider_id": "did:nancy:123-CwDuTckhy2D5VicHSFbVQG",
  "minPrice": 40,
  "maxPrice": 120,
  "duration": "60",
  "responseTime": "30",
}
```



```
"throughput": "0.75",  
"latency": "32"  
}
```

GetService

Returns the service details given the service ID.

Request: *dlt.Request*

- ``string value`` - Service id.

Response: *dlt.Response*

- ``string value`` - The service in JSON format given the id.

- ``string error`` - The error message if any.

Example of service in JSON format:

```
{  
  "id": "36e6049e9a8546a65344da94969ee9123f9df9e01d7b8ffab0eb4e30e0625a47",  
  "model_type": "nancy_service",  
  "model_version": "0.1.3",  
  "owner": "54f23c05b8c66125c870cd0c89bf7c08978fcc28c0cce126be78c5ad0887ef78",  
  "mspi_id": "org0-nancy-dev",  
  "timestamp": 1725538790,  
  "provider_id": "did:nancy:123-CwDuTckhy2D5VicHSFbVQG",  
  "minPrice": 40,  
  "maxPrice": 120,  
  "duration": "60",  
  "responseTime": "30",  
  "throughput": "0.75",  
  "latency": "32"  
}
```

UpdateService

Updates the attributes of a given service.

Request: *dlt.Service*

- ``string serviceId`` - The service ID.

- ``string json`` - The updated service details in JSON format.

Example of update service in JSON format:

```
{  
  "Latency": "33"  
}
```

Response: *dlt.Response*

- ``string value`` - The service in JSON format.
- ``string error`` - The error message if any.

DeleteService

Deletes a service given the service ID.

Request: *dlt.Request*

- ``string value`` - The service id.

Response: *dlt.Response*

- ``string value`` - The service id.
- ``string error`` - The error message if any.

ListService

Lists services based on the provided filter.

Request: *dlt.Request*

- ``string value`` - The filter to apply for retrieving the list of services.

Example of filter in JSON format:

```
{  
  "latency": "32"  
}
```

Response: *dlt.Response*

- ``string value`` - The list of filtered services
- ``string error`` - The error message if any.

Example of list in JSON format:

```
[  
  {  
    "id": "0355534650b3d3b5fe8d35fcb4a91bf175fab6a50534766738167425294bf5f3",  
    "model_type": "nancy_service",  
  }  
]
```

```

"model_version": "0.1.2",
"owner": "54f23c05b8c66125c870cd0c89bf7c08978fcc28c0cce126be78c5ad0887ef78",
"mspi_id": "org0-nancy-dev",
"timestamp": 1725281251,
"provider_id": "did:nancy:123-CwDuTckhy2D5VicHSFbVQG",
"minPrice": 35,
"maxPrice": 115,
"duration": "60",
"responseTime": "30",
"throughput": "0.75",
"latency": "32"
},
{
  "id": "212dee4384ec40d35c6af8adec5c7dab40cd481f206777c156b14c103f0cf707",
  "model_type": "nancy_service",
  "model_version": "0.1.2",
  "owner": "54f23c05b8c66125c870cd0c89bf7c08978fcc28c0cce126be78c5ad0887ef78",
  "mspi_id": "org0-nancy-dev",
  "timestamp": 1725281240,
  "provider_id": "did:nancy:456-4Efs7QsNAaRNYQRdfu2HAn",
  "minPrice": 30,
  "maxPrice": 120,
  "duration": "60",
  "responseTime": "30",
  "throughput": "0.75",
  "latency": "32"
}
}

```

- Methods to manage search

The data model for search is:

- *consumer_ID*: The ID of the consumer. Type string.
- *status*: The status of the search. Type string. Different types of status:

- `INIT`: Initial state.
- `PRICE`: Price state. Ready to set price.
- `SLA`: SLA state. Ready to set sla.
- `ERROR`: Unrecoverable error was happen.
- `FINISHED`: Search is finished.
- *services*: The result to query services using `__service_query__`. Type List<JSON>
- *pricing*: Price data of the search. Type JSON.
- *sla*: SLA data of the search. Type JSON.

CreateSearch

Generates a new search from a consumer with specified parameters. Each search is assigned a new ID and returns the matched services to the consumer. This function further triggers the process for smart pricing and SLA generation.

Request: *dlt.Request*

- `string value` - A request with the following format. Type: JSON

Example:

```
{
  "consumer_id": "consumer1",
  "service_query": {
    "latency": "32"
  }
}
```

Response: *dlt.Response*

- `string value` - The search object created, containing the list of services and the status of the request.

- `string error` - The error message if any.

Example of list in JSON format:

```
{
  "id": "35a11b669999fbac7ee279d2fdcbe1357c9e4b102a987e5613876e40472e787d",
  "model_type": "nancy_search",
  "model_version": "0.1.2",
  "owner": "54f23c05b8c66125c870cd0c89bf7c08978fcc28c0cce126be78c5ad0887ef78",
  "mspi_id": "org0-nancy-dev",
  "timestamp": 1725609228,
  "status": "PRICE",
}
```

```
"services": [  
  {  
    "id": "828e26f6e0f97967e3bf6e8691cc469179a324c031c8af1491a2f9ceec632832",  
    "model_type": "nancy_service",  
    "model_version": "0.1.3",  
    "owner": "54f23c05b8c66125c870cd0c89bf7c08978fcc28c0cce126be78c5ad0887ef78",  
    "mspi_id": "org0-nancy-dev",  
    "timestamp": 1725360787,  
    "provider_id": " did:nancy:123-CwDuTckhy2D5VicHSFbVQG",  
    "minPrice": 35,  
    "maxPrice": 115,  
    "duration": "60",  
    "responseTime": "30",  
    "throughput": "0.75",  
    "latency": "32"  
  },  
  {  
    "id": "a0aab45fef5f1e6cd23173727b81e6758510f27bd281a2221b69f725a98abbd4",  
    "model_type": "nancy_service",  
    "model_version": "0.1.3",  
    "owner": "54f23c05b8c66125c870cd0c89bf7c08978fcc28c0cce126be78c5ad0887ef78",  
    "mspi_id": "org0-nancy-dev",  
    "timestamp": 1725360776,  
    "provider_id": " did:nancy:456-4Efs7QsNAaRNYQRDfu2HAn",  
    "minPrice": 30,  
    "maxPrice": 110,  
    "duration": "60",  
    "responseTime": "30",  
    "throughput": "0.75",  
    "latency": "32"  
  }  
],
```

```

"consumer_id": "consumer1",
"pricing": {
  "provider_id": "",
  "service_id": "",
  "price": 0
},
"sla": {
  "provider_id": "",
  "service_id": "",
  "price": 0,
  "id": "",
  "consumer_id": "",
  "hash_smartcontract": "",
  "service_description": null
}
}

```

GetSearch

Returns the search details given the search ID.

Request: *dlt.Request*

- *string* value - Search id.

Response: *dlt.Response*

- *string* value - The search object given the id.

- *string* error - The error message if any.

Example of list in JSON format:

```

{
  "id": "35a11b669999fbac7ee279d2fdcbe1357c9e4b102a987e5613876e40472e787d",
  "model_type": "nancy_search",
  "model_version": "0.1.2",
  "owner": "54f23c05b8c66125c870cd0c89bf7c08978fcc28c0cce126be78c5ad0887ef78",
  "mspi_id": "org0-nancy-dev",
  "timestamp": 1725609228,
  "status": "FINISHED",

```

```
"services": [  
  {  
    "id": "828e26f6e0f97967e3bf6e8691cc469179a324c031c8af1491a2f9ceec632832",  
    "model_type": "nancy_service",  
    "model_version": "0.1.3",  
    "owner": "54f23c05b8c66125c870cd0c89bf7c08978fcc28c0cce126be78c5ad0887ef78",  
    "mspi_id": "org0-nancy-dev",  
    "timestamp": 1725360787,  
    "provider_id": " did:nancy:456-4Efs7QsNAaRNYQRdfu2HAn ",  
    "minPrice": 35,  
    "maxPrice": 115,  
    "duration": "60",  
    "responseTime": "30",  
    "throughput": "0.75",  
    "latency": "32"  
  },  
  {  
    "id": "a0aab45fef5f1e6cd23173727b81e6758510f27bd281a2221b69f725a98abbd4",  
    "model_type": "nancy_service",  
    "model_version": "0.1.3",  
    "owner": "54f23c05b8c66125c870cd0c89bf7c08978fcc28c0cce126be78c5ad0887ef78",  
    "mspi_id": "org0-nancy-dev",  
    "timestamp": 1725360776,  
    "provider_id": "did:nancy:123-CwDuTckhy2D5VicHSFbVQG ",  
    "minPrice": 30,  
    "maxPrice": 110,  
    "duration": "60",  
    "responseTime": "30",  
    "throughput": "0.75",  
    "latency": "32"  
  },  
  {  
    "id": "828e26f6e0f97967e3bf6e8691cc469179a324c031c8af1491a2f9ceec632832",  
    "model_type": "nancy_service",  
    "model_version": "0.1.3",  
    "owner": "54f23c05b8c66125c870cd0c89bf7c08978fcc28c0cce126be78c5ad0887ef78",  
    "mspi_id": "org0-nancy-dev",  
    "timestamp": 1725360787,  
    "provider_id": " did:nancy:456-4Efs7QsNAaRNYQRdfu2HAn ",  
    "minPrice": 35,  
    "maxPrice": 115,  
    "duration": "60",  
    "responseTime": "30",  
    "throughput": "0.75",  
    "latency": "32"  
  },  
  {  
    "id": "a0aab45fef5f1e6cd23173727b81e6758510f27bd281a2221b69f725a98abbd4",  
    "model_type": "nancy_service",  
    "model_version": "0.1.3",  
    "owner": "54f23c05b8c66125c870cd0c89bf7c08978fcc28c0cce126be78c5ad0887ef78",  
    "mspi_id": "org0-nancy-dev",  
    "timestamp": 1725360776,  
    "provider_id": "did:nancy:123-CwDuTckhy2D5VicHSFbVQG ",  
    "minPrice": 30,  
    "maxPrice": 110,  
    "duration": "60",  
    "responseTime": "30",  
    "throughput": "0.75",  
    "latency": "32"  
  },  
  {  
    "id": "828e26f6e0f97967e3bf6e8691cc469179a324c031c8af1491a2f9ceec632832",  
    "model_type": "nancy_service",  
    "model_version": "0.1.3",  
    "owner": "54f23c05b8c66125c870cd0c89bf7c08978fcc28c0cce126be78c5ad0887ef78",  
    "mspi_id": "org0-nancy-dev",  
    "timestamp": 1725360787,  
    "provider_id": " did:nancy:456-4Efs7QsNAaRNYQRdfu2HAn ",  
    "minPrice": 35,  
    "maxPrice": 115,  
    "duration": "60",  
    "responseTime": "30",  
    "throughput": "0.75",  
    "latency": "32"  
  },  
  {  
    "id": "a0aab45fef5f1e6cd23173727b81e6758510f27bd281a2221b69f725a98abbd4",  
    "model_type": "nancy_service",  
    "model_version": "0.1.3",  
    "owner": "54f23c05b8c66125c870cd0c89bf7c08978fcc28c0cce126be78c5ad0887ef78",  
    "mspi_id": "org0-nancy-dev",  
    "timestamp": 1725360776,  
    "provider_id": "did:nancy:123-CwDuTckhy2D5VicHSFbVQG ",  
    "minPrice": 30,  
    "maxPrice": 110,  
    "duration": "60",  
    "responseTime": "30",  
    "throughput": "0.75",  
    "latency": "32"  
  }  
]
```

```

    "id": "ae91f68b7b543645e83c2e7487d31f6f12572612298b0ff852651b84010fee4d",
    "model_type": "nancy_service",
    "model_version": "0.1.3",
    "owner": "54f23c05b8c66125c870cd0c89bf7c08978fcc28c0cce126be78c5ad0887ef78",
    "mspi_id": "org0-nancy-dev",
    "timestamp": 1725538784,
    "provider_id": "f17c521890ec74a5990b96923bbf7b5635b82ebf5e14c84edcabfcf21d04c86e",
    "minPrice": 40,
    "maxPrice": 120,
    "duration": "60",
    "responseTime": "30",
    "throughput": "0.75",
    "latency": "32"
  },
  {
    "id": "d59fcc44d847f6d5f5ed1cc58a20ea4cd82d83e254722661ae3964f89c047d3a",
    "model_type": "nancy_service",
    "model_version": "0.1.3",
    "owner": "54f23c05b8c66125c870cd0c89bf7c08978fcc28c0cce126be78c5ad0887ef78",
    "mspi_id": "org0-nancy-dev",
    "timestamp": 1725360797,
    "provider_id": "did:nancy:123-CwDuTckhy2D5VicHSFbVQG ",
    "minPrice": 40,
    "maxPrice": 120,
    "duration": "60",
    "responseTime": "30",
    "throughput": "0.75",
    "latency": "32"
  }
],
"consumer_id": "consumer1",
"pricing": {

```



```
"provider_id": "f17c521890ec74a5990b96923bbf7b5635b82ebf5e14c84edcabfcf21d04c86e",
"service_id": "828e26f6e0f97967e3bf6e8691cc469179a324c031c8af1491a2f9ceec632832",
"price": 86.0606208
},
"sla": {
  "provider_id": "did:nancy:123-CwDuTckhy2D5VicHSFbVQG ",
  "service_id": "828e26f6e0f97967e3bf6e8691cc469179a324c031c8af1491a2f9ceec632832",
  "price": 86.0606208,
  "id": "186",
  "consumer_id": "consumer1",
  "hash_smartcontract": "0x2424353",
  "service_description": [
    {
      "id": "",
      "model_type": "",
      "model_version": "",
      "owner": "",
      "mspi_id": "",
      "timestamp": 0,
      "provider_id": "did:nancy:123-CwDuTckhy2D5VicHSFbVQG ",
      "minPrice": 35,
      "maxPrice": 115,
      "duration": "60",
      "responseTime": "30",
      "throughput": "0.75",
      "latency": "32"
    }
  ]
}
```

GRPC API for SLA Operations

The following JSON object represents an SLA (Service Level Agreement) information with various attributes:

```
{
  "id": "unique-sla-id",
  "value": "sla value in json format",
  "provider_id": "provider-id",
  "consumer_id": "consumer-id",
  "provider_sig": "provider signature",
  "consumer_sig": "consumer signature"
}
```

The `id` of the SLA and the `value` field are assigned by the Digital Agreement Creator component, where `value` is a serialized JSON object of the SLA contract.

After the provider or consumer signed the SLA, their signature is saved in `provider_sig` and `consumer_sig`. When one entity is PQC-enabled, the signature field saves the PQC signature; otherwise, it saves the transaction ID that triggers the SLA signing action. This is sufficient as the transaction itself is already signed by the entity.

GetSLA

Returns the SLA content given the SLA ID.

Request: *dlt.Request*

- ``string value`` - The SLA id.

Response: *dlt.Response*

- ``string value`` - The SLA information in JSON format.

- ``string error`` - The error message if any.

GetSLAByConsumerId

Returns the list of SLAs corresponding to the consumer ID.

Request: *dlt.Request*

- ``string value`` - The consumer id.

Response: *dlt.Response*

- ``string value`` - The list SLA information in JSON format.

- ``string error`` - The error message if any.

Slasign

Signs the SLA given the SLA ID using the identity corresponding to uid. The wallet first retrieves the SLA on the blockchain given the `slaid`. If the wallet is registered as a UE, it produces a PQC signature (needs to call external token to get the signature) on the SLA and sends the PQC signature as a transaction payload to the blockchain; if the wallet is non-UE, then the wallet simply invokes `signSLA` transaction to the blockchain. The smart contract verifies the signatures and if successful, and stores the signature (UE) or txId (non-UE) to the SLA entry. Note that for PQC signature verification, the PQC pubkey must first be registered to DIDRegistry.

Request: `dlt.Request`

- ``string value`` - JSON string of the following format where `uid` is the signer and must match either the `ProviderId` or the `ConsumerId` of the SLA referred by `slaid`:

```
{
  "slaid": "slaid",
  "uid": "userDid"
}
```

Response: `dlt.Response`

- ``string value`` - Empty.
- ``string error`` - The error message if any.

SubscribeToSLAInit

Subscribes to SLA initialization events.

Request: `google.protobuf.Empty`

Response: stream `dlt.DltRecordEvent`

- ``string name`` - The name of the event.
- ``string payload`` - The payload of the event, which contains the SLA information in JSON format.

SubscribeToSLASigning

Subscribes to SLA signing events.

Request: `google.protobuf.Empty`

Response: stream `dlt.DltRecordEvent`

- ``string name`` - The name of the event.
- ``string payload`` - The payload of the event, which contains the SLA information in JSON format.

3.2.3.4 Using WALLETGATEWAY and the command-line tool WALLETCIENT

The usage of WALLETGATEWAY is shown as follows:

```
$ walletGateway -help
usage: [-h] [--uid UID] [-p PORT] [--UE]

optional arguments:
  -h, --help  show this help message and exit
  --uid UID   Id of the user, will serve as the prefix of the DID [default: nancy-uid-123]
  -p PORT,   Port of the gateway service
  --port PORT
  --UE       Role is either UE or Non-UE [default: non-UE]
  --sim      Use PQC simulation library, otherwise using the token
```

Note that role of the wallet can also be defined via the environment variable `export WalletRole=UE` or `export WalletRole=non-UE`.

To start a wallet service, we simply appoint an address for the wallet service to listen on:

```
$ walletGateway --port 5000
```

If the wallet runs as an UE wallet, then we specify it on the command line:

```
$ walletGateway --port 5000 --UE
```

Note that since the wallet service is deployed in a docker container. We first import the docker image of the wallet and then start the wallet gateway service using the docker-compose configuration file:

```
$ docker load -i nancy-nec-wallet_0_1_0.tar
# Edit docker-compose-wallet.yaml for the options of the wallet service
# Start the wallet service
$ docker-compose -f docker-compose-wallet.yaml up wallet-service
```

Once the wallet is running, we can use any tool such as `grpcurl` according to the gRPC methods defined in the section above to ask the wallet to talk to the blockchain. For example, suppose the wallet is running on address `localhost:5000`:

```
# Create a provider
$ grpcurl -v -plaintext -d '{"id": "did:nancy:123-CwDuTckhy2D5VicHSFbVQG", "value":
{"name": "vodafone", "type": "publisher"}' localhost:5000 dlt.DltGatewayService/CreateProvider

# List all providers
$ grpcurl -v -plaintext -d '{"value": {"id": {"$regex": "\\.*\\"}}}' localhost:5000 dlt.DltGatewayService/ListProvider
```

To make this communication easier, we further provide a command-line tool `WALLETCLIENT`, and we present the usage of this tool as follows.

```
$ walletClient <action> --help

usage: [-h] -p PORT [-a ADDRESS] [--provider-id PROVIDER_ID]
       [--service-id SERVICE_ID] [--search-id SEARCH_ID] [--sla-id SLA_ID]
       [--consumer-id CONSUMER_ID] [-w] [JSON_STR]

required arguments:
  -p PORT, --port PORT      Port of the wallet gateway service

optional arguments:
  -h, --help                show this help message and exit
  -a ADDRESS,               Address of the wallet gateway service
                           [default: localhost]
  --address ADDRESS
  --provider-id PROVIDER_ID provider id supplied to create a service or
                           get/delete/update a provider
  --service-id SERVICE_ID  service id supplied to get/delete/update a
                           service
  --search-id SEARCH_ID    search id supplied to get a service
  --sla-id SLA_ID          SLA id supplied to get an SLA
  --consumer-id CONSUMER_ID Consumer id supplied to get an SLA
  -w, --wait4event         wait for SLA event after issuing the request

positional arguments:
  JSON_STR                  Input in json string
```

We must first specify an action:

```
Must first specify an action:
```

```
createProvider | getProvider | listProvider | updateProvider | deleteProvider |
createService | getService | listService | updateService | deleteService |
createSearch | getSearch |
getSLA | getSLAByConsumerId | signSLA | listenSLAEvent
```

For example, we can use the `WALLETCLIENT` to create a provider and create a service referring to the previous provider.

```
# The provider starts his wallet on his machine (executable)
$ walletGateway -port 5000 --uid provider1

# The provider starts his wallet on his machine (docker container, update configuration docker-compose-
wallet.yaml for the uid)
$ docker-compose -f docker-compose-wallet.yaml up wallet-service

Wallet service of NonUE starts listening on port 50000...

>>>> Non-UE Wallet <<<<

DID: did:nancy:provider1-FBPWEWXzwu26ZKnxWKX1Wh
EnrollmentId: nancy-provider1-FBPWEWXzwu26ZKnxWKX1Wh
Successfully enrolled user admin and imported it into the wallet

Successfully enrolled user 'nancy-123-FBPWEWXzwu26ZKnxWKX1Wh' with role 'non-UE' and imported it into
the wallet

Server started, listening on port 50000...

[WARNING] Using default DID of the wallet gateway 'did:nancy:provider1-FBPWEWXzwu26ZKnxWKX1Wh' as
provider_id

Subscription request to SLA Signing event.

Subscription request to SLA Init event.

# The provider registers itself on the blockchain, the provider ID is by default set as the DID 'did:nancy-provider1-
FBPWEWXzwu26ZKnxWKX1Wh' of the wallet gateway service running on localhost:5000; otherwise must be
specified via option --provider-id.
$ walletClient createProvider --address localhost --port 5000 '{"name":"vodafone", "type":"operator"}'

# Or
$ docker-compose -f docker-compose-wallet.yaml run wallet-client java -cp wallet.jar WalletClientKt
createProvider --address localhost --port 5000 '{"name":"vodafone", "type":"operator"}'

# The provider now registers a service
```

```

$ walletClient createService -w --address localhost --port 5000 --provider-id 'did:nancy:123-
CwDuTckhy2D5VicHSFbVQG' '{"minPrice": 150, "maxPrice": 250, "Duration": "43", "responseTime": "30",
"throughput": "0.75", "latency": "15"}'

# Or
$ docker-compose -f docker-compose-wallet.yaml run wallet-client java -cp wallet.jar WalletClientKt createService
-w --address localhost --port 5000 --provider-id 'did:nancy:123-CwDuTckhy2D5VicHSFbVQG' '{"minPrice": 150,
"maxPrice": 250, "Duration": "43", "responseTime": "30", "throughput": "0.75", "latency": "15"}'

# Now wait for SLA notification...and you will be asked if agree to sign

```

Then create a search from the consumer side.

```

# The consumer starts his wallet on his machine
$ walletGateway -p 60000 --UE --uid consumer1

# Or with docker container, update configuration docker-compose-wallet.yaml for the port and uid
$ docker-compose -f docker-compose-wallet.yaml up wallet-service

Wallet service of UE starts listening on port 6000...

>>>> UE Wallet <<<<<

number of slots: 1

PKCS#11 Simulator

number of keys: 1

PQC PubKey:
1A89B4896DF55A4BD1F857D7ACBECD23C49AF8E230091D205BADDFEAEC7C63C49398A67C51F6219F52
201531BD62F17E602977143A5605200B7A0223B09DC6633CE1A2F6979710C6B93EF92BC4C81644DF30230
C2A0EE6F387BF84A89E847ED7369C2E090B16C46640EF9A579E3ECD06D1DFCB5B4FBC4507F9016E08B8
9A798197DC2686978152EEF5934BFBE05479B1CA9D50333FEFF686D1B0C5162D5F0D598F086C3B88DF75
7F3B6E40B98BA226F402ACFF684878D84862C7E2FE7CFB3E517C8FE2D9DFCF4DA5246D09BF21ABB487B
DC183...

DID: did:nancy:consumer1-8vu6EbxKhiBUauNFZMUMH4

EnrollmentId: nancy-consumer1-8vu6EbxKhiBUauNFZMUMH4

An identity for the admin user 'admin' already exists in the wallet

Successfully enrolled user 'nancy-consumer1-8vu6EbxKhiBUauNFZMUMH4' with role 'UE' and imported it into
the wallet

Server started, listening on port 6000...

[WARNING] Using default DID of the wallet gateway 'did:nancy:consumer1-8vu6EbxKhiBUauNFZMUMH4' as
consumer_id

Subscription request to SLA Init event.

Subscription request to SLA Signing event.

```

```
# Now the customer can search for the service by providing his consumerId and the searching criteria
$ walletClient createSearch -w --port 6000 '{"consumer_id": "consumer1", "service_query": {"throughput": "0.75"}}'

# Or
$ docker-compose -f docker-compose-wallet.yaml run wallet-client java -cp wallet.jar WalletClientKt createSearch
-w --port 6000 '{"consumer_id": "consumer1", "service_query": {"throughput": "0.75"}}'

# Now wait for SLA notification...and ask if agree to sign
```

After the consumer creates a search, the marketplace will trigger the smart pricing component and digital agreement contract component to create a draft of SLA and register the draft contract in the blockchain (SLARegistry). Then users listening to the *SLAInit* event will get informed and get asked to sign the SLA if they wish. Signed SLA will also emit the event to inform the relevant parties.

3.3 Further Mechanisms for Ensuring the Security and Privacy of the Users

Privacy-preserving Attribute Based Credentials (p-ABC) are digital certificates that allow the owner to disclose only the minimum information necessary to prove his access right to a resource or service. For every resource, a policy defines what attributes are necessary to be proofed to permit access to it. For instance, the policy could demand proof the requester is over certain years old, e.g. “over 18 years old? yes or no”. By using p-ABC certificate, the owner could prove the possession of the property without disclosing the actual value of the attribute. ABC frameworks require certain actors (user, issuer, verifier, revocation authority, registry, inspector). This authentication and authorization (AA) scheme is omitted on the current mobile networks. But its integration is feasible and empowers with privacy the 6G perception. Currently, AA of 5G is based on a shared secret between the user and the system, kept on both sides in a tamper-proof environment, the USIM, and in the Subscription Database of the mobile network. Therefore, subscriptions belong to a certain mobile network operator. To use other mobile network services, the roaming procedure enables remote authentication to users, where the shared secret never leaves the home operator. Only derived keys are provided to the serving network for performing the security of the access stratum (AS) and non-access stratum (NAS) protocols.

In NANCY, we design the integration of p-ABCs in future networks, the subscriber does not need to belong to any network operator, but instead may purchase “attributes” to access different services, including cellular networks. The issuer will grant the user with a p-ABC credential, which later can be used to authenticate against the service providers. Thanks to the use of the p-ABC technology, this process can be carried out in a privacy-preserving way, particularly against the network the user is accessing. Enabling advanced privacy-preserving techniques is a fundamental need for the incipient designs for 6G networks.

p-ABC framework requires that all participants in the framework actors (user, issuer, verifier, revocation authority, registry, inspector) provide their public key in a registry, which may be a decentralized ledger such as a blockchain, in line with current decentralization and self-sovereignty trends. All parties will have access to this information so that they can retrieve the corresponding public keys of the other participants in the communication, if required.

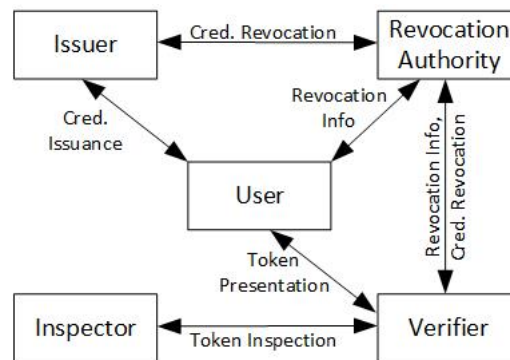


Figure 30: ABC entities and interaction

The relationship between all p-ABC entities is shown in Figure 30. Note that, in specific instantiating of an p-ABC system, some of the roles, such as inspection and revocation authority, may be assumed by the same entity. The user retrieves a credential from the Issuer, who is also exchanging information with the Revocation Authority to set the revocation information for the credential. The User can use the credential to generate presentation tokens for the Verifier, who can ask the Inspector for token inspection and the Revocation Authority for the revocation information. The credential can be revoked if e.g. it is misused, expires or some parameters are exceeded.

The prerequisite for network access is the retrieval of suitable p-ABC credentials from the p-ABC Issuer after the onboarding procedure, i.e. devices can connect to an onboarding network for authentication and secure connection setup as well as the secure provisioning of the long-term credentials. The devices are initially attached to an onboarding network with default credentials and then retrieve their long-term credentials in the form of privacy-preserving Attribute Based Credentials from a provisioning server. The p-ABC credentials can be seen as a digital identity of the device that can be used for different purposes such as authenticating itself in untrusted sub-networks or deriving proofs of its attributes that might be needed to access to net-apps, 6G services, an even authenticating and access directly to other previously unknown devices (as long as they also trust the Issuer) in a D2D interaction.

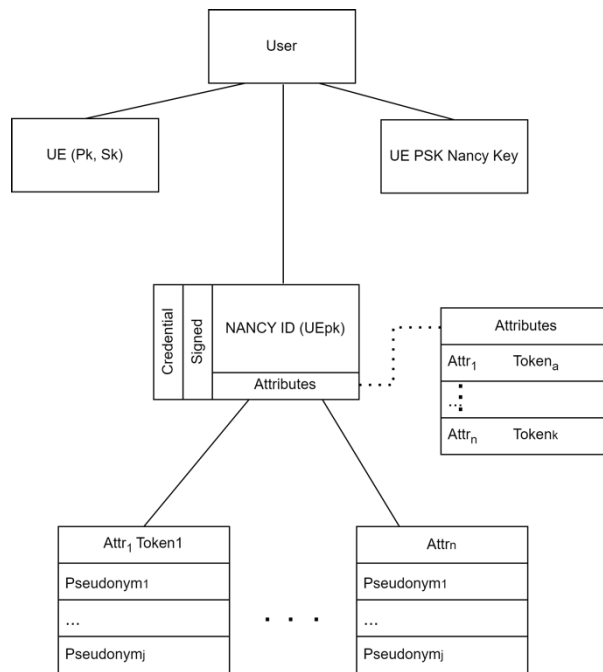


Figure 31: NANCY ID Key hierarchy and Pseudonym generation

To achieve selective linkability, a user may generate a pseudonym from the attributes embedded in the credential during the presentation of the credential to the verifier, i.e., the service owner. In our setting, the device may generate an unlimited number of pseudonyms from the public key as shown in Figure 31. Particularly, scope-exclusive pseudonyms present an interesting alternative, as they can be reused for scopes based, e.g., on the verifier identity and timing data, while remaining unlinkable with any other pseudonym generated by the user.

We now explain how ID management is applied within the NANCY ecosystem to enable privacy-preserving access to third-party services. Figure 32 illustrates the process of credential issuance and access requests to third-party services.

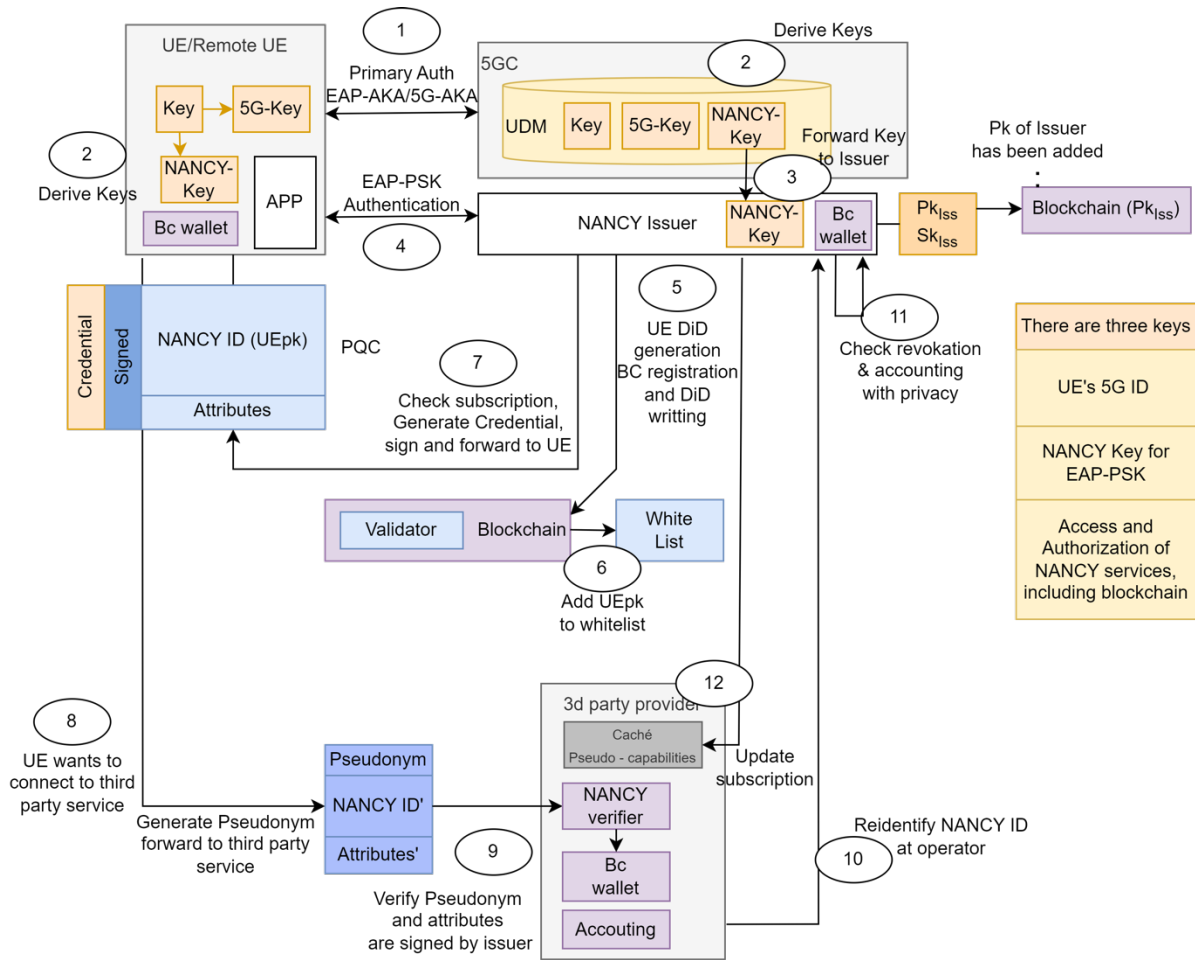


Figure 32. NANCY p-ABC ID management workflow

To retrieve the credential, the UE must first connect to the Data Network. This step is accomplished using a regular 5G connection with the home operator (Step 1). Utilizing a 5G connection is an ideal approach since a shared secret is inherently embedded in the system between the SIM card and the 5G Core. Thus, the keys used for 5G authentication can be leveraged to derive a NANCY key for PSK authentication in an autonomous manner (Steps 2-3). Following this, EAP-PSK authentication takes place between the UE and the Credential Issuer (Step 4).

If the process involves bootstrapping credential generation, the Issuer generates the UE's Distributed Identifier (DID), registers it for blockchain access, and writes the UE's DID onto the blockchain (Steps 5-6). The Issuer then checks the UE's subscription and extracts the relevant attributes. It generates the credential by incorporating the UE's public key (Pk) and packaging the extracted attributes. The credential is signed using the Issuer's private key (Sk) and forwarded to the UE (Step 7). At this point, the credential generation process is complete.

Let us explain how the credential is used to access third-party services while preserving privacy. When the UE wants to access a third-party service, it first requests the service provider to specify the attributes required for authorization. The service provider may request one or more attributes, such as "Access to offloading services," "Access to video processing," or "Access to Nomadic Connectivity Provider" (see Deliverables 4.1 or 4.3). The UE then performs a mathematical operation to transform the original attribute into a pseudonym (Step 8). Pseudonyms conceal the real identity of the UE but

allow the verifier to confirm that the attribute corresponds to a derivative of the requested attribute and that it has been signed by the Issuer (Step 9). This ensures that the UE has the appropriate credentials to access the service.

To achieve this, the third-party provider must have a verifier to handle authentication. The verifier retrieves the Issuer's Pk to validate the signature. Additionally, before granting access to the service, the request with the pseudonym is forwarded to the Original Issuer (Step 10). The Issuer checks for revocation and initiates an accounting mechanism (Step 11). The updated information is returned to the service provider, which can use a caching mechanism to streamline future access for the same pseudonym (Step 12).

4 Research on Additional Security Mechanisms

Section 4 provides a summary of important research that has been carried out in the framework of NANCY, covering (1) the protection of Smart Contracts, (2) Lightweight clients and (3) the impact of decentralization on security.

4.1 Protection of Smart Contracts

Smart contracts with Turing-complete languages have introduced new attack vectors to blockchains. In addition to traditional vulnerabilities such as integer overflow and missing access control, studies [33] [34] [35] on Ethereum smart contract security have shed light on new types of vulnerabilities in smart contract programming such as exception handling, transaction-ordering dependency, reentrancy and front-running. However, with the emergence of new popular blockchains, it is challenging for smart contract programmers to catch on with the awareness of prominent vulnerabilities in those blockchains.

To answer this, we investigated smart contract programming in Solana blockchain [36]. With its launch in 2020, Solana has quickly grown to become one of the most widely used platforms for deploying decentralized applications (DApps) and non-fungible tokens (NFTs). In fact, Solana is among the top 10 blockchain platforms in terms of total market capitalization. Unlike Ethereum’s monolithic smart contract execution environment (the EVM), Solana decouples the execution logic from the state of the smart contract and relies on the programming language Rust. Although Rust poses certain challenges for developers, its security features have been praised and increased developers’ confidence during development. With the Solana execution environment, new vulnerability patterns were introduced that are not captured by existing EVMbased analysis tools [37]. One of the most recent major hacks, the Wormhole hack [38], resulted in the loss of 325 million USD due to a missing key check issue. In particular, attacks exploiting missing ownership and signer checks (that exploit the lack of validation to bypass access control), and cross-program invocation (that exploits the lack of call target validation during cross account calls), seem to be quite intrinsic to the very nature of the Solana execution environment. Compared to Ethereum, there is a lack of understanding of why these vulnerabilities exist, how Solana smart contract developers handle security, what challenges they encounter, and how this affects the overall security of the ecosystem.

Therefore, we set forth to understand the challenges faced by developers during the development of Solana smart contract and to explore the following research questions:

- **RQ1:** Do Solana smart contract developers recognize prominent security vulnerabilities in smart contracts?
- **RQ2:** What challenges do developers encounter that impact the development of secure smart contracts?
- **RQ3:** Given these challenges, what is the prevalence of vulnerabilities in Solana smart contracts?

To address these research questions, we first conducted a developer study, which sheds light on how the developers of the Solana ecosystem handle security and the corresponding challenges they face. Our study comprised a 90-minute code review study with 35 participants (from Upwork, Freelancer.com and Solana Community Insider) and follow-up interviews with a subset of seven participants. We asked participants to write code reviews for a smart contract split into three parts,

each involving one of the most common [34] types of security vulnerability: MSC (Missing Signer Check), IB (Integer Bugs), and ACPI (Arbitrary Cross-Program Invocation). The smart contracts that contain those vulnerabilities are all relatively small and we added multiple non-security-related issues in the smart contract (distraction tasks) to make the review task more realistic and security issues less obvious. Moreover, we set the participants into two groups (A and B) and presented the files in different orders to them.

Table 8. Vulnerabilities and non-security-related issues (distractions) of the review task

File	Vulnerability Type	Line number		Number of Distractions	Lines of Code
		A	B		
<code>item.rs</code>	MSC	53–77	84–108	2	120
<code>tokens.rs</code>	IB	95 & 126	95 & 156	1	140
<code>partner.rs</code>	ACPI	52	73	3	60

Participants assessed their knowledge about these vulnerabilities and explained what challenges they pose in the context of securing smart contracts. Our analysis showed that none of the participants spotted all vulnerabilities in the code review tasks despite their claimed confidence in addressing them; and almost 83% of participants have approved vulnerable code for release. Additionally, participants referred to a shortage of qualified Solana developers, leading to the hiring of inexperienced individuals. The lack of documentation, code reviews, audits, testing, and the complexity of Rust were reported as the main challenges for developers, leading them to adopt alternative frameworks such as Anchor [39].

Table 9. Number of participants found distractions and security vulnerabilities

Vulnerability type	File	Distractions		Vulnerabilities	
		# Found > 0	# Found all	# Found > 0	# Found all
MSC	<code>items.rs</code>	3	1	4	4
IB	<code>tokens.rs</code>	3	3	4	0
ACPI	<code>partners.rs</code>	2	0	2	2
MSC & IB	<code>items.rs</code> & <code>tokens.rs</code>	2	1	2	0
MSC & ACPI	<code>items.rs</code> & <code>partners.rs</code>	1	0	1	1
IB & ACPI	<code>tokens.rs</code> & <code>partners.rs</code>	1	0	1	0
MSC & IB & MSC	all three	1	0	1	0

Table 10. Code review outcomes

Vulnerability Type	File	Releaseable	Releaseable with minor risks	Not releaseable
MSC	items.rs	11 {8}	12 {10}	12 (4)
IB	tokens.rs	14 {11}	12 {10}	9 (4)
ACPI	partners.rs	14 {9}	10 (1)	11 (1)

*The numbers in round brackets show the answers of those who found a valid vulnerability for that file, while numbers in curly brackets exclude the five participants that misunderstood the task.

Given the results of the developer study, one would expect that it would lead to a highly vulnerable ecosystem in Solana. However, recent studies [40] showed that only 52 (0.8 %) projects were found to be vulnerable in Solana. To further confirm those results, we built a framework using symbolic execution to detect the Arbitrary Cross-Program Invocation (ACPI) vulnerabilities in currently deployed Solana smart contracts. These vulnerabilities were among the most challenging to detect by developers, according to our study. Using our tool, we then automatically analysed all 6,324 smart contracts deployed on Solana. Fortunately, our findings corroborate the results [40] and show that only 14 (0.2 %) deployed smart contracts are vulnerable to ACPI.

To conclude, our analysis suggests two conflicting results. On the one hand, Solana developers do not seem to have domain expertise in addressing security challenges when developing smart contracts. On the other hand, the prevalence of security vulnerabilities is fortunately not severe in current Solana smart contracts. Our developer study suggests, however, that the popular Anchor framework provides a healthy tooling environment for developers and is probably one of the most important reasons for the low prevalence of vulnerabilities in the Solana smart contract ecosystem. In fact, our analysis shows that more than 88% of existing Solana smart contracts are currently developed with the help of the Anchor framework without direct native Rust support (see Figure 33 for the trends of using Anchor framework for smart contract development since 2021).

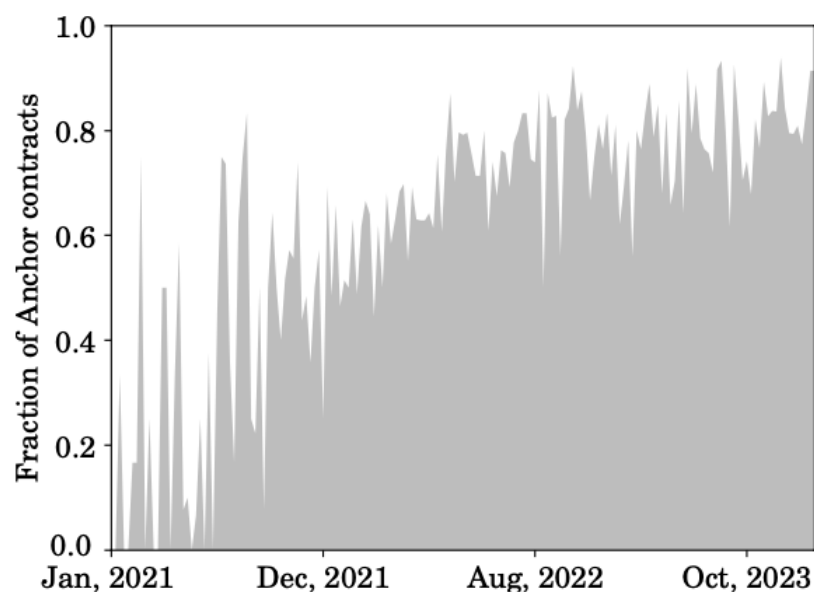


Figure 33 Prevalence of Anchor contracts in all the deployed contracts Solana smart contracts.

4.2 Lightweight Clients

A blockchain-based distributed ledger essentially consists of an append-only data structure replicated across network nodes. Because the ledger keeps growing, storing a copy of the blockchain comes at relatively high costs in terms of computation, memory, and bandwidth. A client who wishes to interact with a blockchain system must either bear the costs of downloading, storing, and verifying the entire blockchain history, or forgo the security guarantees of the blockchain and rely on third-party intermediary servers. The resources required to interact with a blockchain might be prohibitive for some users: for instance, IoT devices are lightweight by design and might not have sufficient memory and CPU for blockchain storage and validation; similarly, typical mobile users have limited bandwidth and may be unable to maintain an up-to-date copy of the blockchain.

This problem was already recognized in the original Bitcoin whitepaper which also proposed the concept of *light client* as a possible remedy and a corresponding realization called Simple Payment Verification (SPV) [1]. Loosely speaking, light clients allow lightweight devices to interact with a blockchain network without storing a local copy of the whole blockchain, while allowing to preserve as much security as possible [41].

More specifically, a light client protocol between a client and one or more full nodes (i.e., nodes which store a copy of the blockchain but do not necessarily participate in the consensus process) allows the full node(s) to share the current blockchain state with the client. Clearly, a solution in which a full node would share the whole blockchain would be computationally unfeasible for most clients. It is hence crucial that the protocol is efficient and minimizes communication, computation, and storage overhead for the client (and ideally also on the full node). In practice, light client protocols typically have full nodes that provide a digest of the current blockchain state along with a cryptographic proof that convinces the client of the correctness of such state.

Several solutions for blockchain light client implementations have been proposed within the blockchain space. For instance, the Bitcoin community provides various implementations for the SPV method. In contrast to verifying full blocks, an SPV light client verifies the chain of proof of work solutions based only on the block headers; to verify that a given transaction is valid, it needs to interact with an honest full node (i.e., additional trust is assumed). Although an SPV light client uses much fewer resources than a full node, its communication complexity is linear in the number of block headers to be verified, which is still impractical for computationally weak environments such as mobile or browser-based clients.

Modern blockchain systems, including Ethereum, are shifting away from Bitcoin's proof of work model towards more environmentally friendly designs, where block generation is regulated through proof of stake or other virtual resources. In these systems, block confirmation is ensured when enough members of an elected committee have signed the block. A similar approach to Bitcoin's SPV light client can be applied to these blockchains, though it comes with the same limitations. Recent proposals for light clients aim to optimize the (linear) communication complexity of this basic SPV design by leveraging additional trust assumptions or offloading computation onto full nodes. Among these proposals, the Proof-of-Proof-of-Stake (PoPoS) design stands out as the most communication-efficient light client implementation in a proof-of-stake context, requiring only $O(\log(m))$ messages to verify m consecutive block headers [42].

A closer look at state-of-the-art light client solutions for committee-based blockchains, such as the PoPoS protocol, reveals that while the improvements significantly enhance the basic SPV design, most light client systems are designed with long offline phases in mind, catering to scenarios where the

number m of consecutive block headers to be verified is quite large (i.e., in the order of weeks). In a setting in which clients can connect to the blockchain with relatively high frequency, solutions designed to operate for large offline periods may be an overkill. For instance, empirical analysis of Bitcoin shows that over a third of light client requests query only a single block (approximately 10 minutes behind), while more than half ask for updates within about 2 hours. Moreover, these protocols often assume that the block headers being verified are signed by continuously rotating committees. While some proof-of-stake designs indeed do use rotating committees (e.g., Ethereum [43] and Algorand [44]), most permissioned blockchains including Hyperledger Fabric as well as some committee-based permissionless systems (e.g., Cosmos [45] and Polkadot [46]) have nearly static committees (i.e., changes in the committee are rare). Therefore, there is an opportunity to leverage the reduced complexity of an almost static committee, in contrast to the case of a rotating committee, to optimize the light client protocol.

Novel light client protocol for committee-based blockchains

As part of the research activities of the project, novel techniques were explored to improve the efficiency of light client solutions, particularly in the context of permissioned systems which are more suitable for industrial deployments. A novel light client design was proposed to specifically balance efficiency, security, and low trust assumptions in blockchain deployments that rely on committee-based blockchains where validator sets are generally stable. This design is particularly suitable for blockchain deployments based on Hyperledger Fabric or variants thereof, as is the case for the NANCY blockchain. The proposed light client protocol is specifically designed to minimize computation and communication costs for both clients and full nodes in committee-based blockchain systems, in the case of (i) a static quorum of validators and (ii) clients being rarely offline for more than a week (henceforth we refer to such operational scenario as “the common case”).

In a committee-based blockchain, the set of validators in charge of validating blocks (i.e., a “committee”) may change over time, however, changes in the committee may only be triggered at pre-determined times called “end of epochs”. In other words, a committee is guaranteed to remain static within an epoch. Consider m consecutive epochs in which a client was offline. When the client is online again, during epoch $m+1$, it interacts with a full node to verify the latest signed digest of the blockchain state. A light client protocol specifically describes this interaction, and it should ensure that upon completion, the client is convinced about the current blockchain state.

A strawman solution would have each epoch’s committee sign end-of-epoch blocks containing the next epoch’s committee and require the full node to share these signatures with the client. Then, a client who knows the committee of epoch 1 and then goes offline for epochs could later be convinced of the current blockchain state by (i) receiving a quorum of validators’ signatures for each epoch, so that she can determine the current epoch’s committee, and (ii) receiving a digest of the blockchain state signed by a quorum of the current epoch’s committee. This strawman solution would imply a communication and computation overhead of $O(m)$ (for receiving and verifying a constant number of signatures per epoch).

Leveraging the peculiarities of the “common case”, the light client solution developed within the NANCY project can greatly improve the communication and computation costs of the strawman solution outlined above.

Briefly, the proposed design requires validators to sign epoch blocks (i.e., blocks indicating the end of an epoch) using a special signature scheme, the so-called transitive signature, that later allows untrusted full nodes to efficiently prove to clients that a given subset of validators in a committee remained static. The gist of the idea is that using this special signature scheme, the signatures

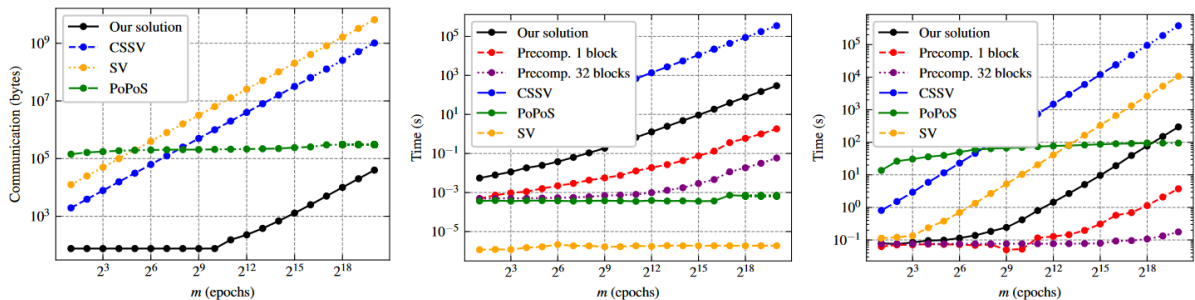
generated by the validators in the static subset of the committee can be aggregated over subsequent epochs. As a result, instead of verifying $O(m)$ signatures as would be the case for a committee being replaced in each epoch, the client only needs to verify $O(1)$ in the best case of a committee being static over the m epochs. The actual light client protocol can offer constant communication and computation at the client also in the “common case” of a “relatively static” committee: indeed, only a static quorum and not the whole committee is required for the client to verify the blockchain state.

Putting it all together, the proposed light client protocol exhibits communication complexity which is asymptotically linear in the number of periods where a quorum of validators remains stable.

Performance comparison with state-of-the-art light client protocols

The performance improvements of the proposed light client solution were demonstrated by means of empirical evaluation and comparison with two state-of-the-art proposals, PoPoS [42] and CSSV [47], in terms of proof size, proof creation time and end-to-end update latency.

A selection of the results is summarized in the Figure 34:



(a) Proof size. For PoPoS: size of all full nodes’ messages to the client. (b) Proof creation time (measured at the full node). (c) End-to-end update latency (measured client-side).

Figure 34. Novel light client protocol: performance evaluation.

To compare the performance of the protocols in scenarios reflecting different clients’ behaviours, the plots depict measured performance as the number of epochs m (during which the client is offline) increases. Notice that while the term “epoch” denotes a specific timeframe, the value of such timeframe depends on the specific blockchain implementation.

As the proposed solution was designed for committee-based blockchains in which the committee does not change drastically over time, in the experiment the length (in epochs) of the period during which a static quorum is present was set to 2000 based on best practices. In such scenarios, the results confirm that the proposed light client protocol outperforms state-of-the-art solutions in terms of both proof size and end-to-end latency, for the “common case” of a client being offline for a relatively short time ($m < 2^{18}$).

4.3 Impact of Decentralization on Security

Pioneered by the Bitcoin cryptocurrency demonstrating the feasibility of realizing Internet-scale digital payments in a decentralized manner, blockchain platforms today provide a general approach to deliver decentralized services. The primary goal of pursuing decentralization is to minimize the reliance on a few trusted central entities: intuitively, removing single points of failure enhances the resilience of the system to attacks. In other words, decentralization is expected to improve the security of the system.

The main technical innovation introduced by Bitcoin is arguably its underlying consensus protocol, the so-called Nakamoto-style consensus, providing the first realization of distributed consensus in a permissionless system (i.e., among anonymous participants who do not know each other's identities). Nakamoto-style consensus provides a general approach for designing permissionless consensus protocols and has indeed been adopted in several other blockchain systems after Bitcoin.

Due to its fundamental significance and practical relevance, Nakamoto-style consensus has received in-depth scrutiny, from both academics and practitioners, leading to numerous theoretical analyses and empirical studies evaluating its security and resilience under different attacker models. The general design of Nakamoto-style consensus includes a probabilistic leader-election protocol (e.g., PoW or PoS) and a method to resolve potential forks, such as the longest chain rule. Intuitively, these two ingredients together guarantee liveness, i.e., new blocks are included in the blockchain at a regular pace, and consistency, i.e., all honest nodes agree on the same sequence of blocks, as long as the majority of resources (computing power in PoW and stake in PoS) is held by honest nodes. To be more precise, Nakamoto-style consensus only provides probabilistic guarantees, meaning that consistency and liveness are guaranteed to hold with a certain probability depending on several operational parameters, and it additionally requires that the underlying network is synchronous, i.e., messages sent across the network must be delivered within a known timeframe.

Several academic studies have formally analysed and refined the necessary and sufficient conditions for achieving security in Nakamoto-style consensus, thereby improving our understanding of permissionless blockchain designs. However, these approaches only partially reflect the original concept of decentralization, as they fail to consider important aspects of how the scale of the network (i.e., the number of nodes n) affects the security of the system. Specifically, while it is commonly understood that an increase in n leads to higher network delays (i.e., larger Δ), the exact impact of these delays on security remains unclear. On the other hand, increasing decentralization naturally leads to enhanced security, as individual nodes hold less relative power. However, current security models typically assume that adversarial power remains unaffected by n , overlooking the beneficial effect of network scale in reducing adversarial influence. A further shortcoming in prior studies is that they abstract away various peculiarities that impact real-world blockchain deployments. For example, simply assuming a synchronous network (as is the case in existing analyses) neglects that the gossip protocols deployed in different Nakamoto-style blockchains, e.g., Bitcoin and Ethereum classic, provide different levels of robustness to adversarial attacks.

Within the research activities of the project, an analytical and empirical investigation was conducted to study how the security of Nakamoto-style consensus is affected by relevant operational parameters: the number of nodes in the network, the maximum network delay, and the relative power controlled by the adversary. Below we summarize the main outcomes of this research.

Extended security model and theoretical analysis

To investigate the impact of decentralization---expressed by the number of blockchain nodes n ---on the security of Nakamoto-style blockchains, a new corruption model was introduced to formally express the intuitive idea that increasing decentralization (i.e., larger n) should make the task of controlling a significant amount of power more difficult for an adversary. While previous security models for Nakamoto-style blockchains assume the relative power controlled by an adversary is upper bounded by a fixed value ρ_{adv} , the proposed model instead allows the adversary to probabilistically corrupt nodes independently of each other and with potentially different corruption probabilities,

capturing realistic scenarios in which, for instance, validators holding large amounts of computing power of stake are better protected and hence harder to corrupt.

Using the newly introduced corruption model, prior results on the security of Nakamoto-style blockchains were extended to capture the intuition that higher decentralization can also have a positive impact on security. Concretely, prior security analyses were extended to provide a more accurate relationship between the number n of nodes in the system, the maximum network delay Δ , and the relative adversarial power ρ_{adv} , the main outcomes being that increasing n causes a logarithmic increase in the maximum delay Δ , and on the other hand, it also decreases the probability that the adversary can successfully corrupt nodes amounting to a certain relative power. (The exact relation between n and ρ_{adv} is rather complex to be expressed concisely without further context, due to the use of probabilistic bounds and the dependency of various technical parameters, and we refer to the published paper for the details [48]). Overall, the proposed security model and extended analysis allow us to express the two opposite trends of how decentralization can harm and benefit security simultaneously in Nakamoto-style blockchains.

Large-scale evaluation

To validate the results of the theoretical analysis, a further empirical evaluation was conducted by simulating different designs of Nakamoto-style blockchains: Bitcoin [49], Monero [50], Cardano [51], and Ethereum Classic [52]. To establish a precise relation between the number of nodes n and the maximum network delay Δ in each of the considered blockchain deployments, large-scale experiments were conducted simulating message propagation, using the gossip protocols deployed in each blockchain design, in a network comprising hundreds of thousands of nodes. A selection of the results, confirming the logarithmic relation between Δ and n and demonstrating how different gossip protocols can lead to diverse network delays, are shown in Figure 35:

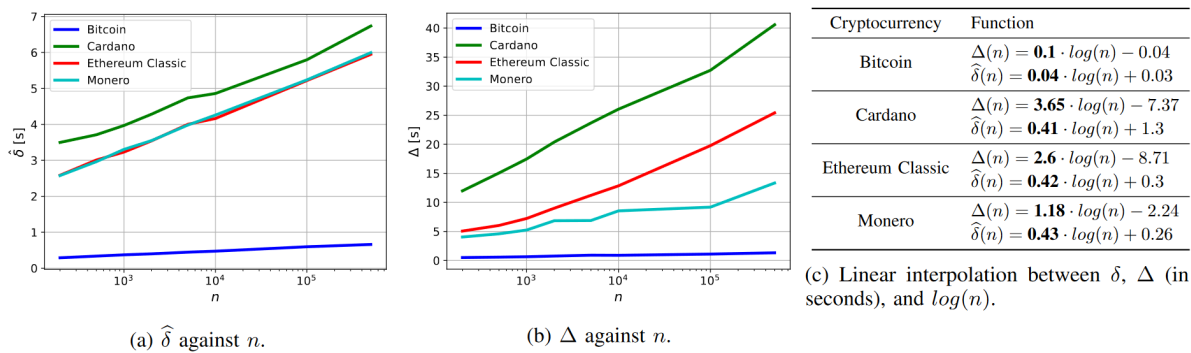
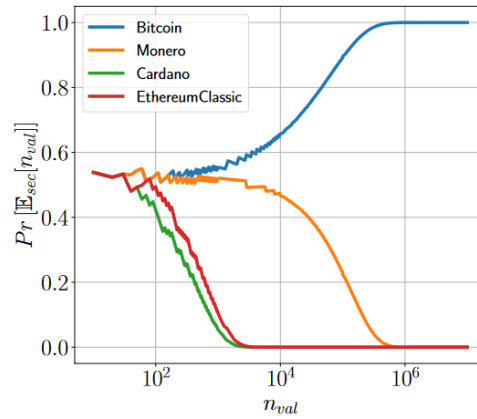


Figure 35: Impact of the number of nodes on the network delay, for different blockchain designs.

The empirically measured delays for increasing n can further serve as a tool to evaluate how the network scale n concretely affects the overall security of the system. Towards this end, one can combine the empirical results on the network delay with the probabilistic analysis evaluating the probability that an attacker corrupts sufficient resources to break security, again varying the number of nodes n . The results, illustrated in Figure 36, show the relation between n and the probability, over the adversary’s corruption attempts, that the honest parties still hold sufficient power to preserve the security of the system, for each of the considered blockchain designs. The value p^* is a parameter

encoding the average provability of corrupting an individual node. For the selected plot p^* was set to 0.16, for instance, the plots show that Bitcoin and Monero scale better in n compared to Cardano and Ethereum classic, in the sense that security can be preserved for larger values of n (because of the message-propagation delays induced by the respective gossip protocols).



(b) The case of $p^* = 0.16$.

Figure 36: Impact of the number of nodes on the probability of preserving security, in different blockchain designs.

5 Conclusions and Future Work

5.1 Conclusions

Although originally developed for digital currencies, blockchain technology represents a foundational innovation with the capacity to revolutionize data management, security, and transparency across various industries. Its decentralized structure, combined with cryptographic security and immutability, makes it well-suited for applications requiring trust and accountability, including communications and identity management. Permissioned blockchains further extend these capabilities by providing enhanced security features tailored to regulated environments, facilitating secure and efficient data exchange.

From a functional perspective, the NANCY Blockchain and its core components support the inter-operator domain of the NANCY system (refer to D6.1 and D4.1). Specifically, they facilitate service handovers between operators through the secure exchange and digital signing of Service Level Agreements (SLAs). This is done by means of several newly designed smart contracts (the NANCY Marketplace, the NANCY SLA Registry) and other components (Digital Agreement Creator, Smart Pricing component) that interact through Oracles and the NANCY Wallet.

Furthermore, the NANCY system is designed to offer significant privacy benefits to users through Decentralized Identifiers (DID) or similar frameworks. In this context, Self-Sovereign Identity (SSI) mechanisms, as outlined in D5.2, are employed. These mechanisms are materialized through, again, newly designed smart contracts (DIDRegistry and VCRegistry) and their corresponding interfaces in the NANCY wallet. They enable users to create a digital identity that remains independent of their formal public identity, granting them full control over its lifecycle, including generation, attribute binding, and deletion. Additionally, NANCY employs a proof-driven approach and utilizes decentralized management mechanisms atop the NANCY Blockchain to oversee identity-related services, such as registration, management, and cryptographic verification data control.

This report reflects the work of T5.2 and T5.3 and provides a detailed overview of these components, their interfaces, workflows, and their integration within the NANCY Architecture. Additionally, data integrity and scalability—both essential features of NANCY—are also addressed.

5.2 Future work

At the time of submission, the consortium is already working on improving and extending the work of tasks T5.2 and T5.3 so that our developments can be used in all use cases in WP6. Namely, WP6 demonstrators will use the blockchain-based components in different, but interconnected, ways.

Firstly, some use cases will make use of NANCY's SSI capabilities (e.g., Italian Outdoor Scenario). For this, an SSI infrastructure will be deployed including verifier, holder and issuer roles, together with code examples. In addition to this, realistic credentials for each use case will be created.

Other use cases will employ the NANCY blockchain and related components to showcase different uses of the inter-operator domain workflow (e.g., Greek In-lab Scenario). For this, the consortium is investigating new fields which enhance the service and provider descriptions. This would include not only service requirements – as those already in place and reported in D5.2 – but also computing resources and localization. The objective is for the NANCY Marketplace to perform a more powerful search, and to determine the best candidates for service handover using more parameters. This work involves not only enhancements to the NANCY Marketplace intelligence but also to the NANCY Wallet interfaces.

Other work will focus on studying the decentralization of practical information about SLA enforcement. For instance, at the time of writing, the particular endpoint for the client (known after the chosen operator's SO enforces the final SLA) that must be sent to the original operator is only known using the *central* Service Manager. However, in the future, this and other centralized information could be added to the NANCY blockchain, and privately associated to the final SLA.

References

- [1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008. [Online]. Available: <https://bitcoin.org/bitcoin.pdf>.
- [2] D. & T. A. Tapscott, *Blockchain revolution: How the technology behind bitcoin is changing money, business, and the world*, Penguin, 2016.
- [3] M. Pilkington, "Blockchain technology: Principles and applications. In F. X. Olleros & M. Zhegu (Eds.)," in *Research handbook on digital transformations*, Edward Elgar Publishing, 2016, pp. 225-253.
- [4] J. Yli-Huumo, D. Ko, S. Choi, S. Park and K. Smolander, "Where is current research on blockchain technology?—A systematic review.," *PLOS ONE*, 11(10), e0163477., 2016.
- [5] V. Buterin, "A next-generation smart contract and decentralized application platform. Ethereum White Paper," 2014. [Online]. Available: <https://ethereum.org/>.
- [6] Z. Zheng, S. Xie, H. Dai, X. Chen and H. Wang, "An overview of blockchain technology: Architecture, consensus, and future trends.," *IEEE International Congress on Big Data*, pp. 557-564, 2017.
- [7] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance and proactive recovery," *ACM Transactions on Computer Systems (TOCS)*, vol. 20(4), pp. 398-461, 1999.
- [8] B. David, P. Gazi, A. Kiayias, and A. Russell, "Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain," *EUROCRYPT (2), ser. Lecture Notes in Computer Science*, vol. 10821, no. Springer, p. 66–98, 2018.
- [9] N. Kshetri, "Blockchain's roles in meeting key supply chain management objectives.," *International Journal of Information Management*, no. 39, pp. 80-89, 2018.
- [10] P. Zhang, J. White, D. C. Schmidt and G. & Lenz, "Applying software patterns to address interoperability in blockchain-based healthcare apps," *Blockchain in Healthcare Today*, no. 1, pp. 1-11, 2018.
- [11] C. Allen, "The path to self-sovereign identity.," 2016. [Online]. Available: <https://www.lifewire.com/https://www.lifewithalacrity.com/article/the-path-to-self-sovereign-identity/>. [Accessed November 2024].
- [12] M. Nofer, P. Gomber, O. Hinz and D. Schiereck, "Blockchain," *Business & Information Systems Engineering*, vol. 59, no. 3, pp. 183-187, 2017.
- [13] Y. Liu, S. Peng, M. Zhang, S. Shi and J. Fu, "Towards secure and efficient integration of blockchain and 6G networks.," <https://doi.org/10.1371/journal.pone.0302052>, vol. 19, no. 4, 2024.
- [14] The Hyperledger Foundation, "What is Hyperledger Fabric?," [Online]. Available: <https://hyperledger-fabric.readthedocs.io/en/release-2.5/blockchain.html>. [Accessed November 2024].

- [15] The Hyperledger Foundation, "Hyperledger Fabric Model," [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.5/fabric_model.html. [Accessed November 2024].
- [16] The Hyperledger Foundation, "How Fabric networks are structured," [Online]. Available: <https://hyperledger-fabric.readthedocs.io/en/release-2.5/network/network.html>.
- [17] The Hyperledger Foundation, "The Ordering Service," [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.5/orderer/ordering_service.html. [Accessed November 2024].
- [18] Androulaki, E., et al., "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains," *Proceedings of the Thirteenth EuroSys Conference, EuroSys '18.*, 2018.
- [19] Lombardi, L., et al. , "Enhancing Blockchain Security and Privacy in 5G and 6G Networks," *IEEE Access*, vol. 9, pp. 24539-24548, 2021.
- [20] Gorenflo, C., et al. , "FastFabric: Scaling Hyperledger Fabric to 20,000 Transactions per Second," *IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2020.
- [21] G. A. Marson, S. Andreina, L. Alluminio, K. Munichev, G. Karame, "Mitosis: Practically Scaling Permissioned Blockchains.," *Proceedings of the 37th Annual Computer Security Applications Conference (ACSAC '21). Association for Computing Machinery*, 773–783. <https://doi.org/10.1145/3485832.3485915>, New York, NY, USA, 2021.
- [22] The Hyperledger Fabric Foundation, "Certificates Management Guide," [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.5/certs_management.html.
- [23] Fabric, "Hyperledger Fabric Docs - Policies," [Online]. Available: <https://hyperledger-fabric.readthedocs.io/en/latest/policies/policies.html>. [Accessed 09 Jan 2025].
- [24] Fabric, "Hyperledger Fabric Docs -- Private Data," [Online]. Available: <https://hyperledger-fabric.readthedocs.io/en/latest/private-data/private-data.html>. [Accessed 09 Jan 2025].
- [25] Fabric, "Hyperledger Fabric Docs -- Peer channel-based event services # Available services," [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/latest/peer_event_services.html#available-services. [Accessed 09 Jan 2025].
- [26] P. Civit, S. Gilbert, R. Guerraoui, J. Komatovic, A. Paramonov, M. Vidigueira, "All Byzantine Agreement Problems Are Expensive," in *Proceedings of the 43rd ACM Symposium on Principles of Distributed Computing (PODC '24)*, New York, NY, USA, 2024.
- [27] J. Liu, W. Li, G. O. Karame, N. Asokan, "Scalable Byzantine Consensus via Hardware-Assisted Secret Sharing," *IEEE Transactions on Computers*, vol. 68, no. 1, pp. 139-151, 2019.
- [28] H. Caliper, "Hyperledger Caliper," [Online]. Available: <https://hyperledger-caliper.github.io/caliper/>. [Accessed December 2024].

- [29] "Hyperledger Fabric Samples," [Online]. Available: <https://github.com/hyperledger/fabric-samples>. [Accessed December 2024].
- [30] H. F. Docs, "Taking ledger snapshots and using them to join channels," [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/latest/peer_ledger_snapshot.html. [Accessed Feb 2025].
- [31] W3C, "Decentralized Identifiers (DIDs) v1.0 -- Core architecture, data model, and representations," 19 July 2022. [Online]. Available: <https://www.w3.org/TR/did-1.0/>.
- [32] W3C, "Verifiable Credentials Data Model v2.0," 27 Jan 2025. [Online]. Available: <https://www.w3.org/TR/vc-data-model-2.0/>.
- [33] M. Rodler, W. Li, G. Karame and D. Lucas, "Sereum: protecting existing smart contracts against re-entrancy attacks," *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.
- [34] S. Cui, G. Zhao, Y. Gao, T. Tavu and J. Huang, "Vrust: automated vulnerability detection for solana smart contracts," *2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.
- [35] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel and G. Vigna, "SAILFISH: Vetting Smart Contract State-Inconsistency Bugs in Seconds," *2022 IEEE Symposium on Security and Privacy (SP)*, 2022.
- [36] S. Andreina, T. Cloosters, L. Davi, J.-R. Giesen, M. Gutfleisch, G. Karame, A. Naiakshina and H. Naji, "Defying the Odds: Solana's Unexpected Resilience in Spite of the Security Challenges Faced by Developers," *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024.
- [37] Neodyme.io, "Solana smart contracts: common pitfalls and how to avoid them.," 20 August 2021. [Online]. Available: https://neodyme.io/en/blog/solana_common_pitfalls/.
- [38] C. Team, "Lessons from the Wormhole Exploit: Smart Contract Vulnerabilities Introduce Risk; Blockchains' Transparency Makes It Hard for Bad Actors to Cash Out," 30 January 2023. [Online]. Available: <https://www.chainalysis.com/blog/wormhole-hack-february-2022/>.
- [39] A. Ferrante and M. Callens, "Anchor Framework," [Online]. Available: <https://www.anchor-lang.com/>. [Accessed 16 2023].
- [40] S. Smolka, J.-R. Giesen, P. Winkler, O. Draissi, L. Davi, G. Karame and K. Pohl, "Fuzz on the beach: fuzzing solana smart contracts," *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, Copenhagen, 2023.
- [41] P. Chatzigiannis, F. Baldimtsi, K. Chalkias, "SoK: Blockchain Light Clients," *Financial Cryptography and Data Security*, Grenada, 2022.
- [42] S. Agrawal, J. Neu, E. N. Tas, D. Zindr, "Proofs of Proof-Of-Stake with Sublinear Complexity," *5th Conference on Advances in Financial Technologies (AFT)*, Princeton, NJ (USA), 2023.

- [43] E. Foundation, "Ethereum whitepaper," [Online]. Available: <https://ethereum.org/>. [Accessed 23 December 2024].
- [44] Y. Gilad, R. Hemo, S. Micali, G. Vlachos, and N. Zeldovich, "Algorand," Proceedings of the 26th Symposium on Operating Systems Principles. ACM, pp. 51–68, Oct. 14, 2017.
- [45] "Cosmos," [Online]. Available: <https://cosmos.network/>.
- [46] "Polkadot," 2024. [Online]. Available: <https://polkadot.com/>.
- [47] O. Ciobotaru, F. Shirazi, A. Stewart, and S. Vasilyev, "Accountable Light Client Systems for {PoS} Blockchains," Cryptology {ePrint} Archive, 2022.
- [48] J. Albrecht, S. Andreina, F. Armknecht, G. Karame, G. Marson, and J. Willingmann, "Larger-scale Nakamoto-style Blockchains Don't Necessarily Offer Better Security," 2024 IEEE Symposium on Security and Privacy (SP), pp. 2161–2179, 2024.
- [49] "Bitcoin," [Online]. Available: <https://bitcoin.org/en/>.
- [50] "Monero," [Online]. Available: <https://www.getmonero.org/>.
- [51] "Cardano," [Online]. Available: <https://cardano.org/>.
- [52] "Ethereum Classic," [Online]. Available: <https://ethereumclassic.org/>.