THE SCALARIS
DESIGN SPECIFICATION

Table of Contents

Introduction

This document is intended for both a nontechnical and a developer audience, provided the reader has at least a foundational understanding of blockchain technology. The Scalaris design and architecture are introduced in a largely nontechnical way, with the aid of diagrams, and through a step-by-step exposition of what inter-chain services are and thus what an inter-chain infrastructure service must provide at a minimum. With increasing frequency as the document progresses, the discussion turns from what inter-chain services *would* be like to specifications of what the Scalaris *is*. By combining matters of design with those of implementation and integration, the intended result is a systematic design specification of the Scalari.

WHY WE ARE RELEASING THIS DOCUMENT

We believe that there is no better way to design solutions than to communicate and to gain insights on our work from multiple perspectives. We also believe that by not keeping our work private, we stand to gain enormously from people's engagement with it and with our code. Thirdly, in an infrastructural project, it is of paramount importance to engage those who would use and build upon our technology.

Some may be concerned about the perceived risk that competitors could copy our work and gain the advantage, especially because our design is both first-to-market and has yet to achieve major traction. Be that as it may, the combined insights of the crypto and enterprise architecture communities are a resource that we cannot afford to be isolated from, and so the risk-reward ratio leaves us unconcerned.

CONTRIBUTORS WELCOME

The Scalaris is neither a company nor an exclusive team, it is infrastructure, and we believe that infrastructure ought to be publicly owned and freely available to all.

This document is currently a draft and not a final design. In fact, the notion of a final design has no clear place in a project that practices continuous development. We welcome contributions and discussion to this document.

The Scalaris is opensource project anyone may contribute to the Scalaris project.

A project such as this stands to benefit the most when a variety of perspectives and skills come to bear on its design and implementation. We welcome collaboration of every variety.

To contact the Scalaris, email scalariscoin@gmail.com

Opportunity

ABOUT THE SCALARIS

The Scalaris is infrastructure for the coming “inter-blockchain era,” an emerging technology epoch characterised primarily by the superseding of the current API ecosystem with a decentralized and intrinsically monetized “token ecosystem.” This will occur when its enabling technologies (specifically, smart contracts and “dapps”) mature to the point of possessing practical inter-blockchain interoperability. At the time of writing, the Scalaris is one of the technological leader in the provision of inter-chain infrastructure for use by dapps and smart contracts.

We believe that the emergence of the inter-blockchain era will have disruptive implications for *two* sectors, that of software-as-a-service, and practical blockchain usability.

From the perspective of software-as-a-service (SaaS), the token ecosystem embodies two fundamental advancements: (a) the comparatively frictionless monetisation of digital services, and (b) the leveraging of the unique robustness, decentralization, and security properties of blockchain technology.

From the perspective of blockchain technology, if blockchains are to achieve their true potential, then broad, generic interoperability between blockchain services is required. Without inter-chain interoperability, blockchain-based services will (a) either deliver services only within the confines of the limited customer base that runs its nodes, or sacrifice the unique security properties of blockchains in delivering to centralized entities, and (b) face enduring problems with chain bloat and, relatedly, the market-related pressure to build further features onto a single chain.

By creating an “internet of blockchains,” the Scalaris is positioned to enable the frictionless monetisation of APIs, and in doing so, to empower blockchain technology by converting its thousands of isolated chains into a token ecosystem.

PAIN-POINT

Traditional internet-based services are faced with perennial insecurity in their technology stack. Furthermore, they typically require the centralization of functions and data, placing a high trust-burden upon their customers. In contrast, blockchain technologies enable one to exploit cryptographic proofs to deliver “trustless” services, where each participating entity may prove to itself the certitude of a given outcome, and thus radically reduce the amount of trust required to do business with another party. This systematically enlarges the range of ways of doing business, enables many new business models, and may provide clearly defined security guarantees, lowering costs and better protecting brand value.

Yet blockchains cannot immediately achieve their true potential, for the primary reason that they are not interoperable. There are thousands of blockchains in existence, yet they currently function like LANs disconnected from the internet, and have yet to create the circumstances that will foster the era-defining disruption that generic interconnectivity brings – on a scale similar to how the internet enabled the emergence of Facebook and Google.

SOLUTION

The Scalaris is foundational infrastructure for the token ecosystem. It provides true peer-to-peer interoperability between nodes on different blockchains, in order to enable the following:

- The delivery of potentially *any* kind of digital service from a node on any blockchain to another.
- The ability for any given blockchain service to function not as an "appcoin" but as a "protocol service," that is, to be consumable by *any other* dapp on any blockchain, for open-ended purposes, instead of only the purposes of its creators' dapp, greatly enlarging the service's market reach and revenue stream.
- The ability for smart contracts' tokens to function not merely to monetise "dapps" but to be "protocol tokens," logically placing them at a layer lower in the technology stack, where their potential utility is at a greater order of magnitude. Additionally, services' code quality may benefit from a broad contributor-base of developers from diverse communities, exploit their combined learnings, prevent chain bloat and code duplication, save labour-time, and deliver services to the entire blockchain-consuming market, instead of just the set of users of its blockchain.
- The ability for dapps to be simple orchestrations of inter-chain services instead of difficult hand-coded creations from the ground up. The primary development tasks thus become API integrations, not the difficult and highly specialised role of coding new and "bulletproof" smart contracts.
- The building of dapps with a microservices architecture, where each blockchain may deliver a single service, integrated with many others in a modular fashion, providing simpler component design, easier bug fixing, and easier upgrading.
- The ability to effectively bypass the (currently-difficult) matter of choosing which blockchain to build upon – and not only at the start of a project, but at later points in its lifecycle, when various microservices may become better-implemented on a different blockchain.
- The monetisation of inter-chain and multi-chain services, using their intrinsic tokens of value.
- The full exploitation of new, cryptoeconomically-driven business models ushered in by blockchain technology. For example, businesses may extract value from a "better than free" model, from monetary policy directly (ICOs, transaction fees, deflationary economics, block rewards, and superblock self-funding systems), and from a marketplace for its monetized APIs.

The Scalaris shall achieve the above through an architectural and protocol-based approach, the documentation of which is the subject of this paper.

Design

DESIGN OBJECTIVES

The following features shall be designed for, in descending order of priority:

1. Interoperability

First and foremost, the Scalaris is inter-blockchain infrastructure. As such, its most direct design objective shall be interoperability with an overwhelming majority of existing and future blockchain implementations. Additionally, it shall be interoperable with centralized entities in order to make traditional server-based services available within the token ecosystem.

2. Decentralization

To be decentralized is, essentially, for no one entity to exercise control over other entities in a system. For example, perhaps Bitcoin's major achievement is – in broad terms – the decentralization of money, in which no one entity controls (a) the currency's value, (b) the transferral of funds, (c) the keeping of records of account, and (d) its monetary policy.

Yet Bitcoin currently exists in an ecosystem that is largely centralized, nullifying many of its benefits in practice. It is of little value to offer a centralized ecosystem for the delivery of decentralized services, since (a) this exists already, in the form of the API ecosystem, and (b) the property of interest, namely decentralization, would largely be lost during service-delivery. For example, if one buys Bitcoin using a centralized exchange, the purchase is not "trustless" because one must trust the exchange, and the purchase is subject to all the usual friction of traditional payment infrastructure (bank fees and delays, payment gateway fees, visa and mastercard fees, fraud risk, KYC requirements, the requirement to trust many intermediaries with one's money and personal information, and so forth). Hence, in order for Bitcoin and every other decentralized technology to achieve its potential, a decentralized *ecosystem* is required, where entities may do business without compromise to the technologies' disruptive power.

3. Security

Decentralized and monetary services characteristically require high security and high determinacy of operation at a level comparable to aeronautical applications, because (a) it is not generally possible to alter or take offline a service that runs on the edges of its network, on its users' devices, and (b) if money were found to be stealable in a system not subject to central rectification, then it would very quickly lose most of its value. For these reasons, the Scalaris requires the highest level of security and determinacy of operation.

4. Trustless Service Delivery

In the context of blockchains, a frequent and desirable consequence of decentralization is that it is not necessary to trust a counterparty to act honestly over the course of a transaction. For example, with Bitcoin, one does not need to trust a middleman to transfer funds or a recipient to report honestly on whether the payment was received or what its amount is, since no middleman is involved and counterparties may independently verify a payment's status with an extremely high degree of confidence.

In the case of inter-chain service delivery, an equal degree of "trustlessness" is required when payments for services are made between blockchains, in order that the service may be rendered and paid for without requiring participants to act honestly, thus preserving this unique feature of blockchain-based payments in an inter-chain context.

5. Simple Integration (No Coding Required)

To maximize interoperability and to reduce friction, integration to the Scalaris and access to the token ecosystem shall not require modification of stock wallets or nodes. Note that consumption of some third party service which leverages the Scalaris for its delivery *may* require coding, but the use of the Scalaris itself shall not.

6. Decentralized Integration

To maximize security and to foster an open, internet-style ecosystem, integration to the Scalaris and access to the token ecosystem shall not require the mediation of any central entity (even us). To deliver or consume services over the Scalaris, consumers shall not be required to (a) use the Scalaris blockchain, (b) use any specific service, or (c) use any service that has a centralizing effect. (Here "centralizing" is taken to denote a range of scenarios, from control by a central agent to a sidechains-style centralization of networks around its network. The latter we describe as "inter-chain centralized.")

Note that consumption of some third party service delivered over the Scalaris may require the mediation of some central party, but the use of the Scalaris itself shall not.

7. Composability

As far as is possible, the Scalaris shall be built with composability and modularity in mind, in the same pattern in which inter-chain microservices are envisaged above. Specifically, the key principles of microservice design is to maximise composability while being mindful of which services will always be consumed together, in order to avoid building a "distributed monolith." These are preserved unchanged in the context of a token ecosystem.

8. Monetisability

In the token ecosystem, an additional key principle is added to the principle of composability: that a service be intrinsically monetizable. If not, then we suggest it be bundled into a monetizable service's API, or else the people who would run the service's nodes may lack a reason to, since they'd not be able to derive a revenue stream from it.

Furthermore, a service's revenue stream is required to be *secured* via some trustless protocol or via cryptoeconomic incentives, or else value is not likely to be captured. Monetisability is as much a question of whether a consumer of your service will be willing to pay for it as it is a question of whether they are unable to forcibly consume it for free.

The Scalaris shall monetise its core services where feasible, offer others for free, and shall provide various means by which services delivered over the Scalaris may be monetized securely.

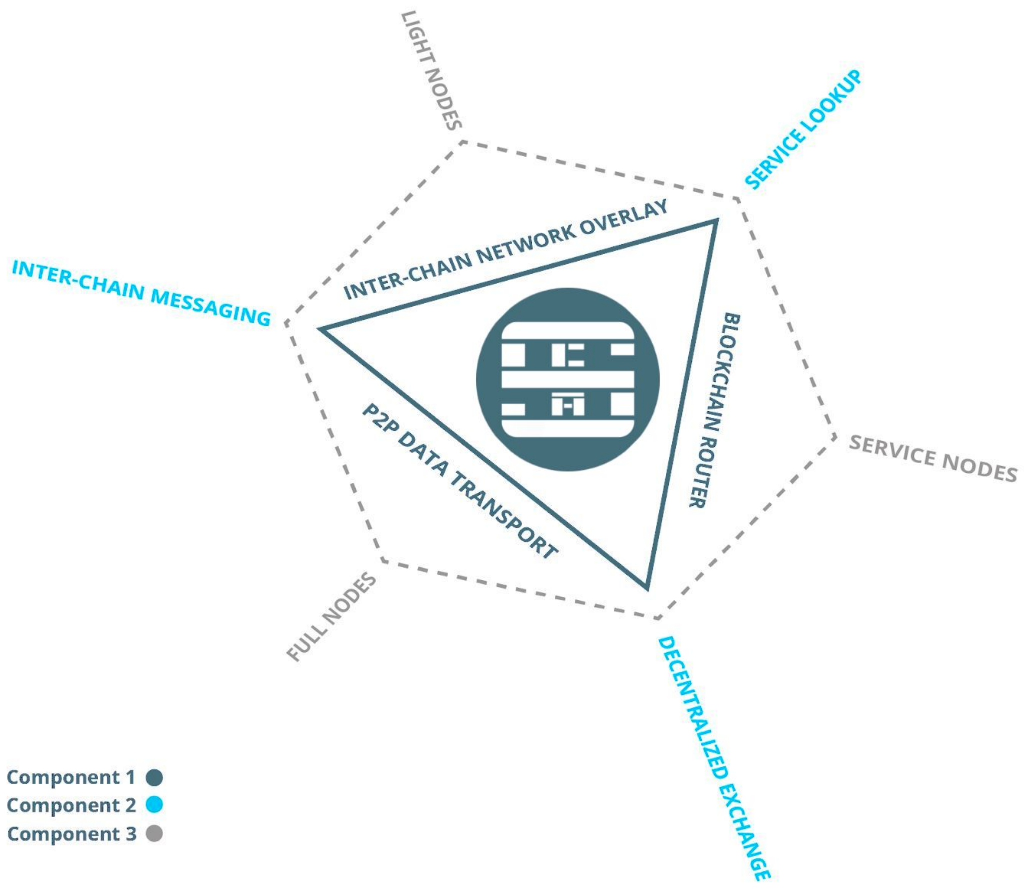
9. Mobility and Small Footprints

Several scenarios, from mobile apps to embedded IoT devices in the insurance, health, supply chain, agriculture, automotive telematics and security industries, stand to leverage the token ecosystem.[1]Many of the use-cases we expect to emerge will require dapps with a very light footprint, which would thus be unable to host even a single blockchain. The Scalaris shall provide access to the token ecosystem for such devices, in order that they may harness blockchain-specific security properties which we feel are mission-critical to reducing the attack surface of IoT services. Specifically, the Scalaris shall enable applications with a small footprint to consume (and pay for) inter-chain services *without* hosting a blockchain locally.

ARCHITECTURE

General purpose inter-blockchain interoperability is achieved by the integration of three **core components**, which together function to deliver three **core services**, accompanied by any number of **blockchain services** and **blockchain components.** These serve to enable the building of an unlimited number of inter-blockchain services – a token ecosystem – all of which may be orchestrated into **inter-chain applications**.

To aid the reader in this novel territory, these shall be introduced with the help of a series of diagrams, which articulate the relations between components and services. The diagrams shall progress from one to the next as follows:



The components shall be introduced first, followed by the services. Prior to this though, the general nature of inter-chain architecture shall be introduced.

What Does Inter-Chain Architecture Look Like?

In general, inter-blockchain architecture will always involve at least two blockchain networks, and some additional entity or function to deliver interoperability between them. Since blockchain networks are decentralized and distributed, the interoperability component(s) ought not be placed in some central location; to preserve decentralization, they are required to either run on, or locally interface with, nodes at the edge of each network.

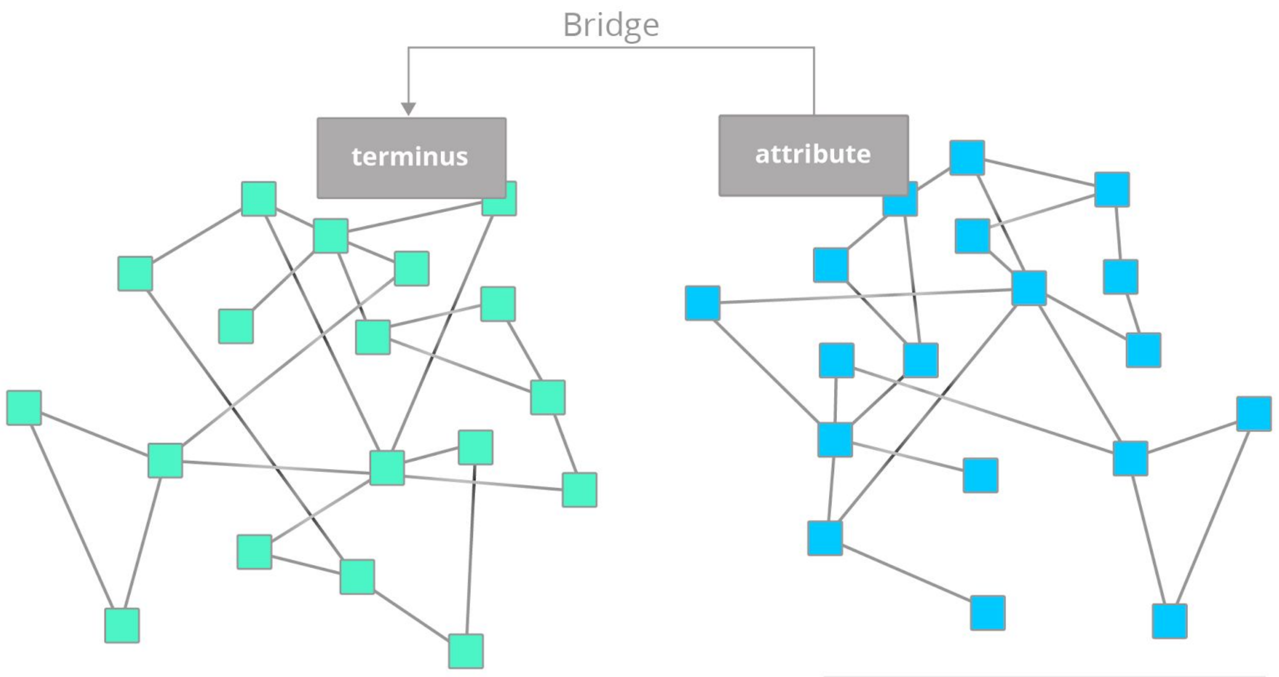


Figure 1
Idealised pair of p2p networks, & disintermediated delivery of a service from one node to another.

Various projects have proposed solutions of the following kinds:

- **traditional technology:** a centralized intermediary (e.g. Poloniex.com)
- **maximalists:** a decentralized *network* which functions, as a logically centralized intermediary (e.g. Bitcoin in a sidechains context)
- **proprietary code** (i.e. wallets, smart contracts, or bolt-ons to wallets) that achieve blockchain interoperability *only* between nodes running this code (i.e. BTCrelay)
- **a walled gardens:** inter-chain protocols only between instances of some custom blockchain, which locks developers into building upon it (e.g. Aion).

None of the above varieties of inter-chain technology are both *generic* and *decentralized*. That is, they either do not provide support for an open-ended variety of services (including those on *existing* blockchains), or they fail to provide such support without centralizing control in a way that betrays a given service's dependency upon being decentralized in one respect or another.

As per the Scalaris **design objectives**, a satisfactory solution must be both generic and decentralized. We approach this by "first principles," that is, by remaining faithful to the nature of the inter-chain scenario itself.

1. Distributed Network Architecture

Firstly, of unambiguous importance is that any inter-chain component(s) must exist on the edges of the networks they interoperate with. This distributes the service across each blockchain network that delivers or consumes services. Additionally though, the inter-chain components must deliver services from the edges of their *own* network too, without requiring central action, or else it will function as yet another centralized intermediary.

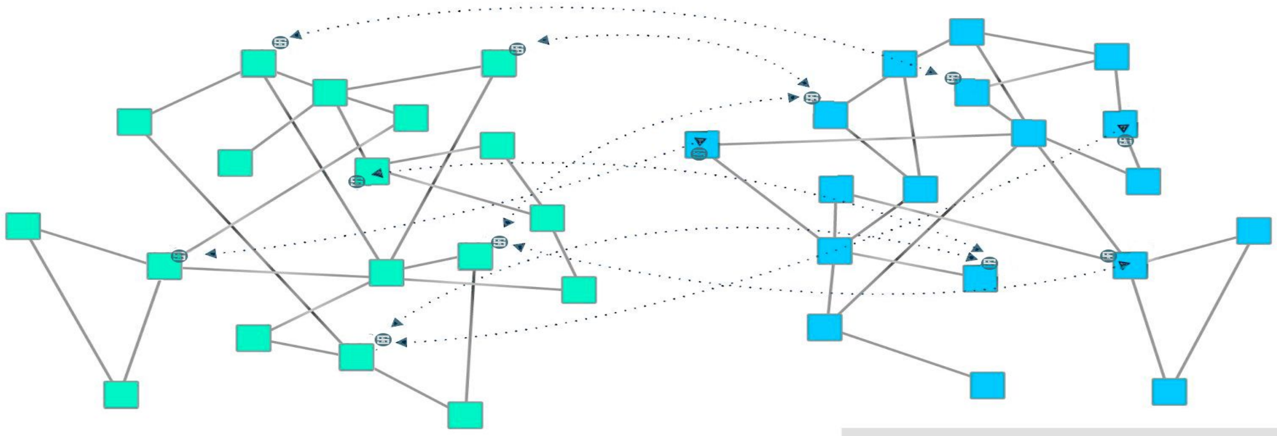


Figure 2
A suitable inter-chain service must exist on the edges of both its network and the network(s) using it.

2. Decentralized Actors

Secondly and relatedly, an act of delivering or consuming an inter-chain service must be self-sovereign, that is, not subject to control by a third party. Architecturally speaking (that is, apart from protocol design), the most direct and secure means to achieve this is for nodes of the inter-chain service component and either the network consuming or delivering it, or both, to *exist on the same local machine*. The extent to which this is necessary – and to which it impacts the footprint of an inter-chain service – will vary, ranging from a requirement to run full nodes, to SPV nodes, to merely signing transactions, through to, at a minimum, querying a blockchain explorer website or other centralized oracle in low-security applications. The latter shall be considered a limit-case for the applicability of the term "inter-chain."

As such, the full range of local architecture requirements is evinced. In every case other than the limit case, some manner of direct participation in both a delivery and a consumption network is required for each actor to participate in a decentralized fashion. This may be graphically indicated by iterating upon the previous diagram:

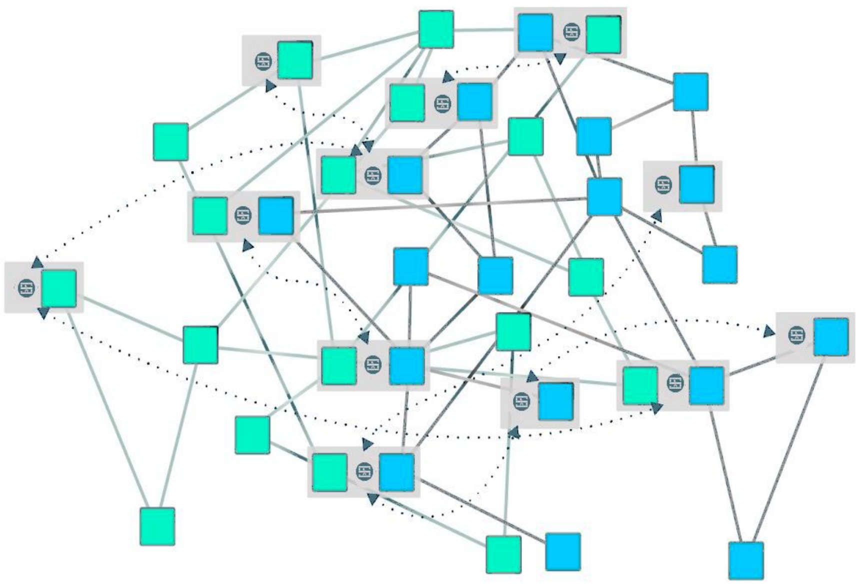


Figure 3
Decentralized Inter-Chain Infrastructure
Gray areas denote varying combinations of consuming, delivering, and inter-chain nodes. (Nodes outside gray areas do not support inter-chain services.)

3. No Blockchain Lock-In

While every inter-chain service must be delivered from nodes on some chain, inter-chain infrastructure must not limit *which* blockchain a given service may run upon, or else all that is achieved is essentially a kind of distributed client-server model, which is in fact the default architecture for centralized applications today. For example, Blockstream's implementation of sidechains would require that every user interact with the Bitcoin blockchain in order to consume any other chain's service. We term this pitfall "inter-chain centralization." To avoid it, a true internet of blockchains – and one that can support a token ecosystem – must enable services to be delivered or consumed from any blockchain.

This chain-agnostic notion motivates for the careful minimising of integration requirements and app footprint. For example, if the Scalaris were to require that every consumer of an inter-chain service maintain a copy of the Scalaris blockchain, possibly in addition to the service provider's blockchain, then its usefulness would be rather limited and its user-friction be very high.

This aspect of inter-chain infrastructure design will come to bear chiefly upon the *monetized* delivery of a service, since, firstly, on a peer-to-peer network, actors are untrusted and it is necessary for payment and service-delivery to be atomic. Secondly, a node must be paid in its native token by a node with a different native token, and so they will have to be exchanged, and decentralized exchange requires a high degree of security and code quality. Yet if it is required that two or even three blockchains are downloaded and maintained in order to consume the service, it is unlikely that it will see widespread adoption. Hence, the Scalaris shall provide means to avoid this.

Summary

The above considerations yield three guiding principles for the design of the Scalaris components:

1. Inter-chain infrastructural services must run on the edges of both *their* network(s) and any service-delivery and consumption networks.
2. Architecturally speaking, service-decentralization is most easily achieved by running components required for either the delivery or the consumption of a service on the same local machine.
3. Inter-chain infrastructural services must limit their integration requirements and footprint where possible.

CORE COMPONENTS

The Scalaris comprises three core components, which function together as the foundation of a general-purpose inter-chain service infrastructure:

- **XBridge**, an inter-chain network overlay
- **XName**, a blockchain router
- **XChat**, a p2p data transport

These three components are defined as "core" because, intuitively, *any* inter-chain interoperability solution will necessarily require some means of networking between nodes on different underlying networks, some means by which nodes may discover where to route service requests, and some protocol for p2p communication once a suitable node is located.

To aid the reader in remembering and visualising the complex of components and services in the Scalaris, a diagram shall be constructed progressively as elements are introduced. The following diagram shows just the Scalaris three core components.

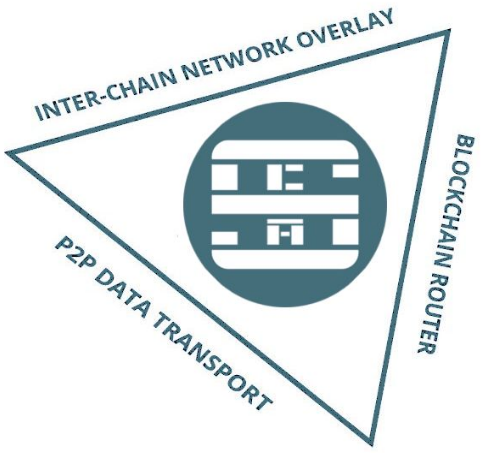


Figure 4
First Iteration: the Scalaris Three Core Components.
The necessity, in order for a triangle to exist, for there to be three sides joined at their apexes, signifies the joint necessity of these components for the delivery of inter-chain services.

XBRIDGE: THE INTER-CHAIN NETWORK OVERLAY

The Scalaris features XBridge, a serverless DHT-based peer-to-peer network. On a given local machine, nodes on this network are integrated with nodes on *other* networks, making our network an inter-chain network overlay. This enables lookup, location, and broadcast between nodes on any blockchain network.

Context Diagram

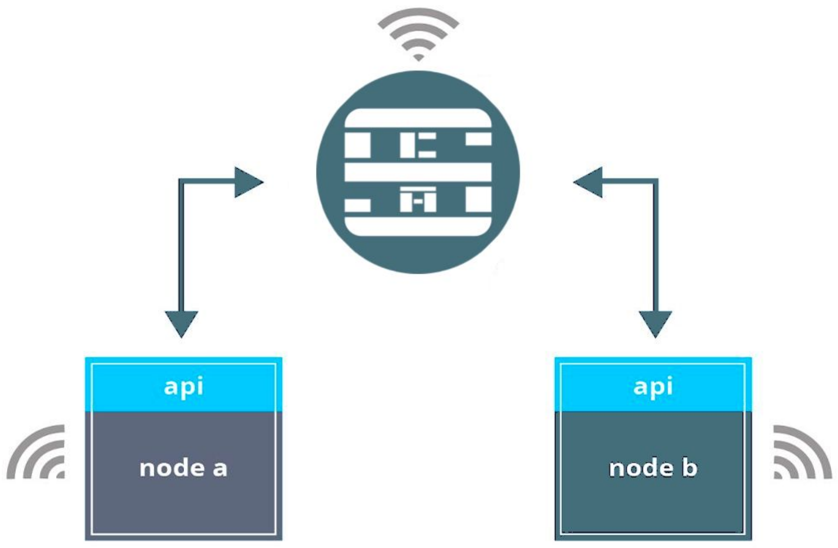


Figure 5
Network components on a local machine.

For technical documentation, refer to the **Technical Specification** section.

Implementation

Current: the network overlay code is implemented in both XBridgep2p.exe and in the Scalaris wallet. It is not monetized.

Future: this may be modularised in the codebase, though it is unlikely to be released as a standalone application, since it will interoperate with other components to deliver **core services**.

XNAME: THE BLOCKCHAIN ROUTER

An ecosystem of inter-chain services requires a means of routing messages to the correct blockchain, which, in the Scalaris, shall be achieved via an inter-chain address system. Blockchain routing is in its elementary stages and requires exploration and agreement by the broader crypto community, but, fundamentally, it requires an inter-chain standard for designating blockchains such as Uport’s MNID, a way of committing routing data to a registry, and a lookup function. A key outstanding question is the optimal price-to-truthfulness ratio of routing results. For example, services may benefit most from a free registry service and tolerate a certain degree of bad lookup results by eliminating the possibility of dishonesty after lookup, immediately prior to payment and delivery. Alternatively, some services may require provably truthful lookup results and would tolerate a microfee for this. XName shall take an agnostic approach to registry service design, and its mature solution shall provide for diverse integrators’ needs if necessary - including the possibility of the emergence of a competitive marketplace for registry services.

As such, a logical router design direction is to offload matters of the truthfulness and cost of (a) lookup and (b) the committing of routing data to dedicated registry services – including lookup of the chainIds of registry services. XName's function would thus be to invoke registry services' APIs and, after service delivery is completed, to store locally a cache of routing results.

To ameliorate circularity (that is, having to look up a registry service before one can look a service up) on initial launch, nodes may bootstrap themselves either (a) querying a hardcoded chainId for a provably truthful registry service, or (b) by querying their peers over XBridge with a dedicated get Registry Service call, and then querying each registry service returned, thus exploiting whatever truthfulness guarantees each service may provide in order to build a truthful local list of registry services (and other inter-chain services). To return to the question of how to commit and look up inter-chain services, see both the **Service Lookup** and the **Registry Service** sections.

For technical documentation, refer to the **Technical Specification** section.

Implementation

Current: the blockchain router is implemented in both XBridgep2p.exe and in the Scalaris wallet, along with the other core services.

Future: this component is likely to always interoperate with other components to deliver **Core Services**, and so will not necessarily be made deployable independently. However, it may be, if sufficient progress is made to render blockchain routing monetizable, since its monetisation will enable the service to reflect its own running costs, to allow competition between routing services, and to incentivise the technological advancement of the service. If monetized, the router component may additionally integrate with other components on their own blockchain network.

XCHAT: A PEER-TO-PEER DATA TRANSPORT

The delivery of digital services requires a means of sending and receiving both messages and the service's payload itself. As such, the Scalaris features XChat, an end-to-end-encrypted, peer-to-peer messaging module that supports both one-to-one messages and group messages. (Broadcast messages are already supported by **XBridge: the Inter-Chain Network Overlay**.)

The requirements for communication and for digital service delivery vary with the nature of the service. Privacy, bandwidth, latency, persistence, and the absence or presence of intermediaries are all variables. As such, this service may mature into several data transport technologies. At this very early stage though, a supremely private, fast, peer-to-peer solution appears to be adequate.

Technical documentation is available in the **Technical Specification** section.

Implementation

Current: the data transport is implemented in both XBridgep2p.exe and in the Scalaris wallet along with the other core services.

Future: this may be modularised in the codebase, and if it is monetizable as a standalone application, may be released as such.

CORE SERVICES

Monetizable inter-chain services require three core infrastructural services:

- **Service Lookup:** a way of discovering peers to deliver or consume a service

- **Inter-Chain Messaging:** a way of delivering a digital service
- **Decentralized Exchange:** a way of monetising the delivery of the service

These services are *orchestrations* of the **Core Components**, and as such may be represented on the apexes of the previous diagram's triangle, as follows:

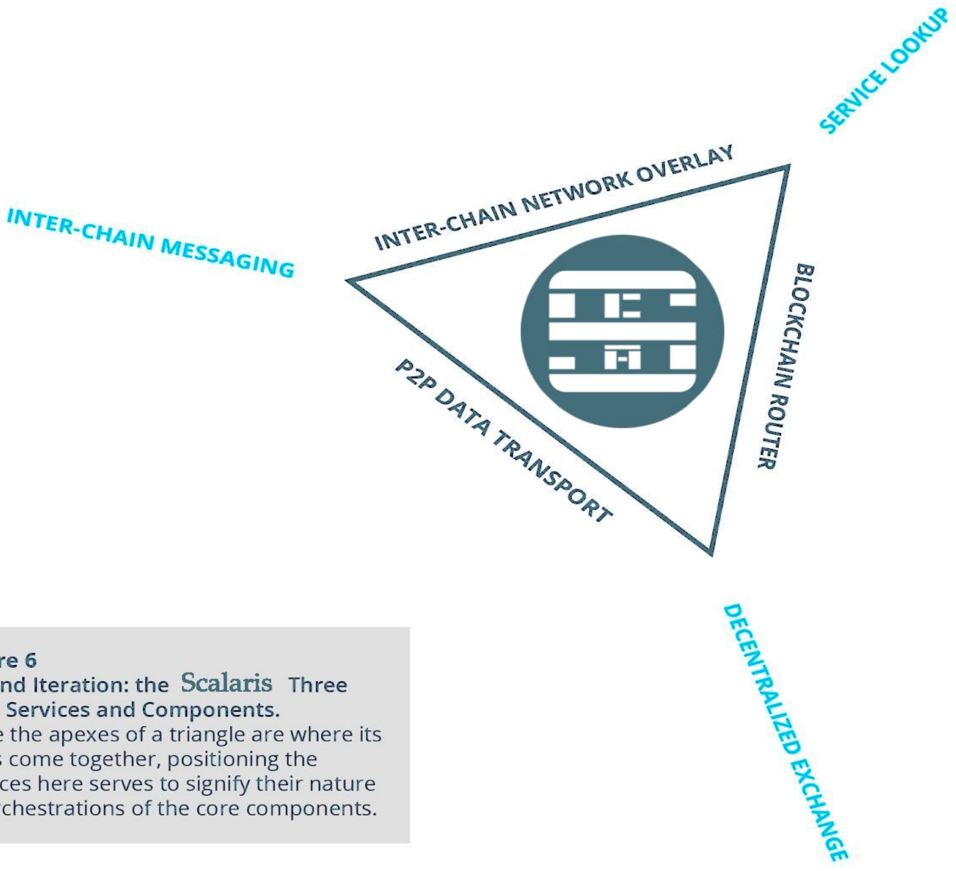


Figure 6
Second Iteration: the Scalaris Three Core Services and Components.
Since the apexes of a triangle are where its sides come together, positioning the services here serves to signify their nature as orchestrations of the core components.

SERVICE LOOKUP

Inter-chain service lookup is an orchestration of the XBridge, XName, and XChat components, and invokes any given **registry of inter-chain services** on the Scalaris. It may be abstracted into an API façade as the Scalaris matures.

Like the traditional internet, parties need to look up and locate each other in order to interoperate. Thus, an analogue of the Domain Name System (DNS) is required. Unlike the traditional internet though, services are delivered peer-to-peer and generally by any node on a network, rather than from servers addressable from a single IP address, and so in order to request a service it will characteristically only be required to locate *any* node on a given blockchain network. Hence, "chain codes" are the primary requirement, along with other secondary properties.

It is assumed that services shall generally be *delivered* to a specific node and to that node only.

A significant feature of a registry service over a peer-to-peer network of untrusted nodes is that, unlike circumstances in which the service provider may reasonably be trusted, *anyone* with any intention (malicious or otherwise) may provide it. Depending on the design strategy, either the truthfulness of the data returned from service lookup shall not be guaranteed, and therefore guarantees as to the service's integrity must be provided at a later stage, or service lookup must be conducted in such a way as to cryptographically guarantee the truthfulness of the data regardless of the intentions of the node providing the service. The Scalaris design is intended to be agnostic as to which approach to registry design is taken.

Message Steps

On the former design assumption, the typical steps for service lookup are:

1. Find peers on XBridge
2. Retrieve a service's chainId and serviceList over XName
3. Query peers over XBridge to find a peer on the chain designated by the chainId
4. Switch to XChat; get a list of the services the specific peer supports
5. Request a service (and proceed to construct a proof of the truthfulness of the service)

On the latter design assumption, service lookup may be trivially accomplished (that is, with no original contribution from this project) by employing a blockchain consensus algorithm (e.g. proof-of-work) and for the node looking up a service to host locally a blockchain containing the chainId data and simply query it for free. But this imposes a significant storage and uptime burden upon a user, and so is not expected to be a typical way of using the Scalaris. Outside of a context where users maintain a service registry blockchain, SPV nodes, as described in the original Bitcoin whitepaper, may be employed. For greater scalability and a smaller app footprint, as of the time of writing, alternative proof systems are being explored. See the **Trade History** and **Registry Service** sections for design patterns.

Context Diagram

The above steps imply the following high-level architecture:

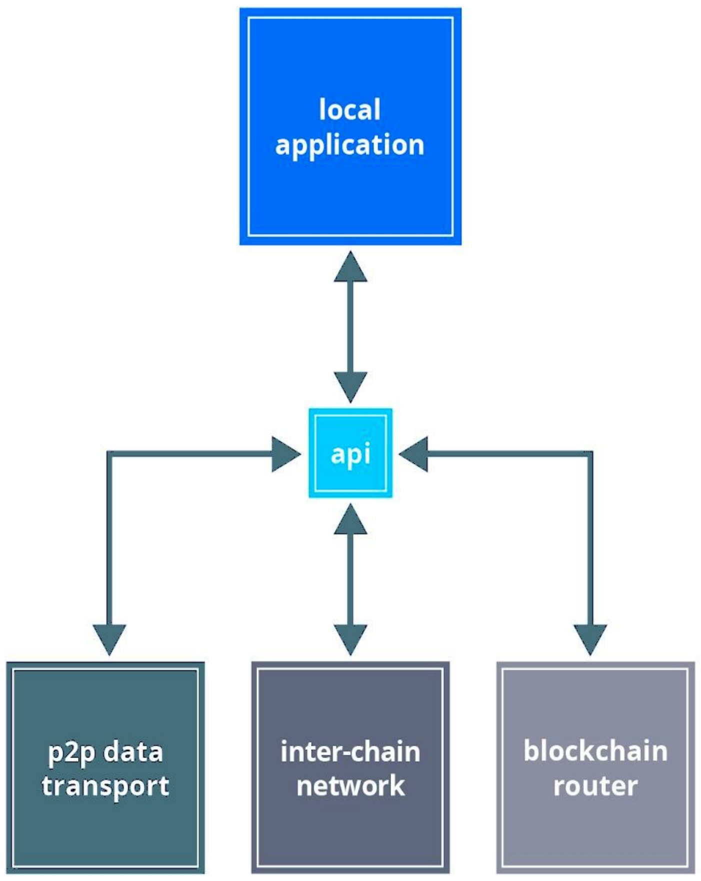


Figure 7
Context diagram indicating relations of core components on a local machine utilising service lookup.

Technical documentation is available in the **Technical Specification** section.

Implementation

> **Current:** Service lookup is currently implemented either as part of XBridgep2p.exe or embedded into the Scalaris wallet. At the Scalaris current stage of development, the only application implemented is a decentralized exchange, which requires only that nodes filter by currency pair. As such, a more general-purpose lookup service remains to be built.
>
> **Future:** Upon suitable exploration and consensus by those in the blockchain ecosystem, development of a generic lookup service is planned. Furthermore, to simplify consumers' integration with the underlying components, service lookup may become embodied in an API façade. Whether this will be a single façade for all the core services, or a distinct façade specifically for lookup, is probably a function of time, maturity, and consumer demand.

INTER-CHAIN MESSAGING

Inter-chain messaging is an orchestration of **XBridge** and **XChat**, and is a matter of a service provider and consumer locating one another, communicating, and the service being delivered.

This service is specified under the assumption that consumers shall actively find services, while service providers shall passively be locatable.

On public peer-to-peer networks it cannot be assumed that every entity presenting itself as a service provider or consumer will act in good faith, and so for most services, it will be necessary to both prove that the service's payload is legitimate prior to making payment, and that the consumer cannot receive it without also making payment. In other words, services must function both *trustlessly*and *atomically*.

A planned way to achieve "trustlessness" in this context is through the use of a zero-knowledge proof system. As the notes on BIP-0199 state, "[v]arious practical zero-knowledge proving systems exist which can be used to guarantee that a hash preimage derives valuable information. As an example, a zero-knowledge proof can be used to prove that a hash preimage acts as a decryption key for an encrypted sudoku puzzle solution. (See pay-to-sudoku for a concrete example of such a protocol.)" Note, for the sake of clarity, that a simple digital signature scheme set up to prove that the service provider is in fact who the consumer thinks it is constitutes a zero-knowledge proof, in the sense that the service provider's private key is not revealed, yet the consumer may prove to itself the identity of the service provider. The actual proof-schemes implemented on one service or another can be expected to vary with the particulars of what needs to be proved for the "trustless" delivery of the service.

Atomicity in this context is achieved through **decentralized exchange**, in the following section.

Message Steps

Typical steps in the inter-chain messaging service are:

1. Find a service over the network overlay and request a service, as per the preceding section
2. Over XChat, the service provider supplies materials for a zero-knowledge proof of the payload's legitimacy – that is, for creating a proof over some set of assertions that sufficiently ensures the payload's legitimacy (see the **Trade History** section for an elementary example)
3. The consumer accepts the service
4. The service provider proceeds to the decentralized exchange service to commence service-delivery

In the interest of minimising app footprints or for simplifying consumption, inter-chain messaging may demand a distinct API façade. Use-cases where this seems pertinent are where services are free and where chain lookup can be hardcoded, for example, a decentralized chat app.

Context Diagram

The above implies the following high-level architecture:

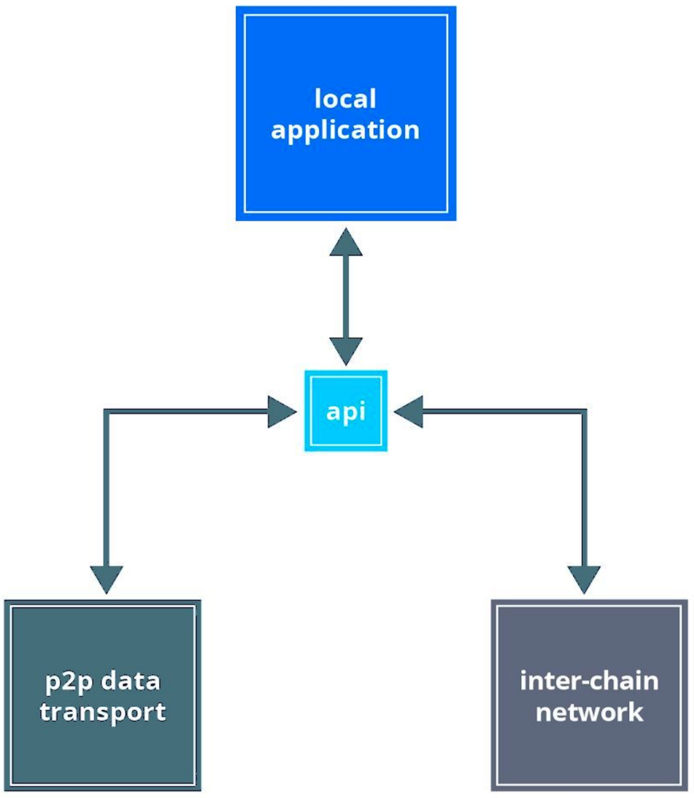


Figure 8
Context diagram indicating relations of core components on a local machine utilising the inter-chain messaging service.

Implementation

Current: Developers may (a) embed **XChat** into their applications, and (b) exploit the network overlay, either through the XBridge standalone application or in its embedded form in the Scalaris wallet.

Future: Components shall be modularised and APIs written, and in addition, potentially an API façade abstracting and orchestrating the two components into an explicitly defined service.

DECENTRALIZED EXCHANGE

Decentralized exchange is a trustless, atomic means of exchanging currency for *either* another currency or for a service. It is an orchestration of **XBridge**, **XChat** and at least two wallets or blockchain nodes in a generalised implementation of Noel Tiernan's atomic exchange protocol. This protocol exploits the creation of a secret, the knowledge of which is required for the spending of funds, *and* which cannot fail to be disclosed when funds are spent. Thus, if its creator spends a counterparty's funds, then the counterparty possesses the secret and may spend the creator's funds in return.

Note: the decentralized exchange service *is distinct from the decentralized exchange* application*, Scalaris DX, which consumes this service in additional to several **blockchain services**.*

Preceding Work

The first version of the protocol was shown to be vulnerable to a malleability-based extortion attack, and later rectified, in the Bitcoin-and-clones ecosystem, with the introduction of the opcode OP_CHECKLOCKTIMEVERIFY. A design utilising the latter is detailed in Kay Kurokawa's blog on the subject. Finally, to support not only atomic swaps of cryptocurrencies and tokens, but also any digital payload, we add to this protocol the use of a decryption key or secure hash function *as* the secret. As such, the revealing of the secret enables the immediate consumption of a previously-supplied encrypted digital good, or the verification of some truth-claim. This, in conjunction with the use of zero-knowledge proofs in the preceding section to allow a consumer to verify the credibility of a digital good in advance of receiving it, achieves *the trustless consumption of monetised goods and services*. As such, it is a *general* monetised service protocol.

Protocol Steps

Conceptually, the protocol steps are as follows:

1. The service provider creates either a decryption key (for digital goods), a digital signature or cryptographic hash of some file (for truth-claim verification), or a random number (for coin trading), and uses it as the protocol's secret.
2. The service provider creates a "bail-in" transaction with the rule "this transaction may either be spent by the consumer if (s)he supplies the secret, or if both the consumer and service provider sign the transaction." (If a digital good is being offered, rather than coins, then only a quantity of coins sufficient to cover a network fee is required.)
3. The service provider creates a fallback mechanism in the form of a second, "refund" transaction, with the rule "send the bail-in transaction's output to the service provider's address, but not earlier than (*x*) amount of time from now."
4. The service provider requests that the consumer sign the refund transaction, in order to meet the second requirement in the bail-in transaction that it may be redeemed if both consumer and service provider sign it.
5. The consumer signs the refund transaction and returns it to the service provider.
    a. *Note: this enables the service provider to reclaim its coins (if any) after the specified time period, in the event that the consumer abandons the exchange or another factor causes it to fail to complete.*
6. The service provider broadcasts the exchange transaction.
    a. *Note: the consumer may now "spend" this transaction if (s)he is given the secret.*
7. The consumer creates her own "bail-in" transaction, as in step 2, for coins on her own blockchain.
    a. *Note: this transaction requires the* same *secret to be revealed in order for it to be spent – which only the service provider is in possession of at this stage.*
8. The consumer creates her own "refund" transaction, as in step 3.
9. The consumer requests that the service provider sign her refund transaction, as in step 4.
10. The service provider signs the refund transaction and returns it to the consumer.
    a. *Note: this enables the consumer to reclaim her coins in the event that the service provider fails to redeem them, since otherwise the secret would not be revealed to the consumer and she would thus not be able to consume the service or receive coins in a trade.*
11. The consumer broadcasts her bail-in transaction
    a. *Note: the service provider may now spend the consumer's coins, since it possesses the secret.*
    b. *Note: by spending the consumer's coins, the service provider reveals the secret publicly, and thus the consumer may now consume the service.*

   c. *Note: if the service provider would prefer to cancel the exchange, it may wait until time* x *in step 3 and broadcast its refund transaction.*

12. The service provider spends the consumer's exchange transaction, revealing the secret.
    a. *Note: if the service provider fails to spend her bail-in transaction, then the consumer may broadcast her refund transaction and receive the coins back after the specified time period.*
    b. *Note: the service provider must spend the consumer's bail-in transaction within the time period specified in her refund transaction, or else the consumer is free to refund herself.*
13. The consumer uses the secret either to spend the service provider's bail-in transaction, or to decrypt a file, or to verify a fact.
    a. *Note: in the case of a coin trade, the consumer must spend the service provider's exchange transaction within the time period specified in its refund transaction, or else the service provider is free to refund itself.*

Note that the above protocol shall include additional supplementary steps in which **Service Nodes** are updated as to the state of the atomic swap, in order for SPV nodes and "light" clients, which depend upon their peers to relay transactions and messages, to be reliably updated.

Message Steps

As implemented in the Blocknet's components, decentralized exchange typically involves the following steps:

1. After the consumer accepts a service as per the steps in the **Inter-Chain Messaging** section, the service provider takes protocol steps 1-3, and then, over XChat, step 4
2. The consumer takes protocol step 5 over XChat
3. The service provider takes protocol step 6 over its native blockchain network
4. The consumer takes protocol steps 7-8, and then, over XChat, step 9
5. The service provider takes protocol step 10 over XChat
6. The consumer and service provider take protocol steps 11-13 over their respective native blockchain networks

Note that these steps only concern the decentralized exchange *service*. The decentralized exchange *application*, Scalaris DX, like most applications, requires several additional (that is, non-core) services in order to function as an exchange, specifically, order broadcast, order matching, anti-spam measures, anti-DOS measures, and trade fee collection.

Context Diagram

The above steps imply the following relations between components on a local machine:

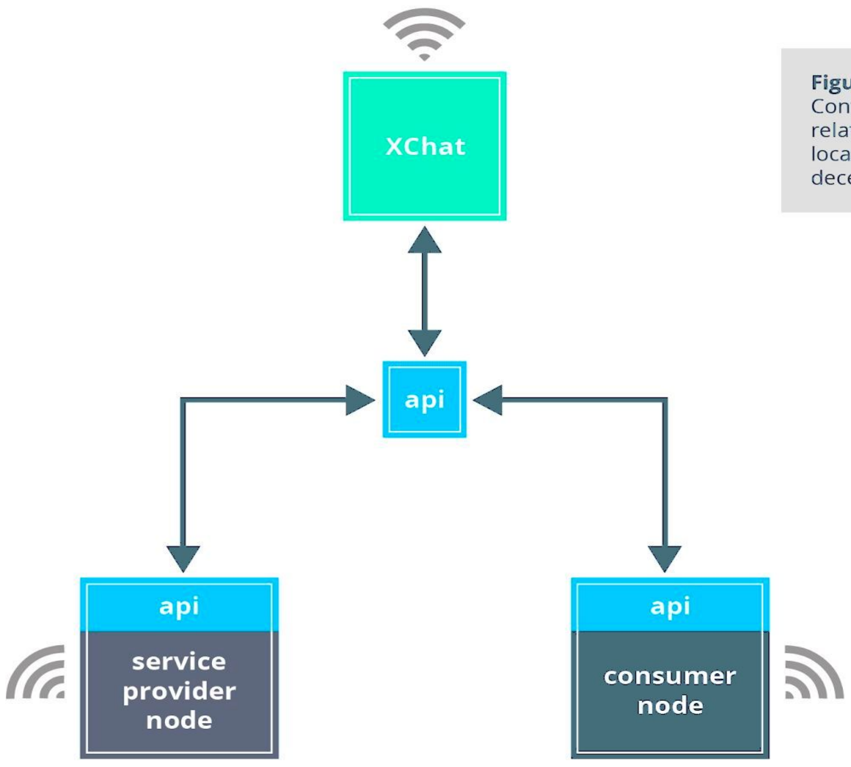


Figure 9
Context diagram indicating relations of components on a local machine utilising the decentralized exchange service.

Implementation

Current: The decentralized exchange service is embedded in the Scalaris wallet, and in the standalone XBridgep2p.exe (however, development will resume on the latter at a later stage). An API is available, concurrently supporting the decentralized exchange *application*.

Future: The service shall be abstracted from the various blockchain services that Scalaris DX consumes, and an API façade built to orchestrate XChat and the blockchain services.

BLOCKCHAIN COMPONENTS

On a given local machine, the three core services introduced in the preceding section may interact with any one or a combination of the following component-types, which may be on several blockchains:

1. **Full Nodes:** "regular" full-featured nodes and wallets
2. **Light Nodes:** SPV nodes and those which are even lighter (e.g. those that merely sign transactions)
3. **Service Nodes:** nodes with special features to provide a given service over and above regular blockchain work

These component types are generally third-party integrations not built or maintained by the Scalaris. Nonetheless, they provide a necessary function in the Scalaris, namely interacting with their native blockchains, without which the Scalaris would not achieve interoperability without duplicating it on its own components – which would be an untenably inefficient approach.

The three core components and services, plus any additional node-types consuming or delivering an inter-chain service, are thus representable as follows:

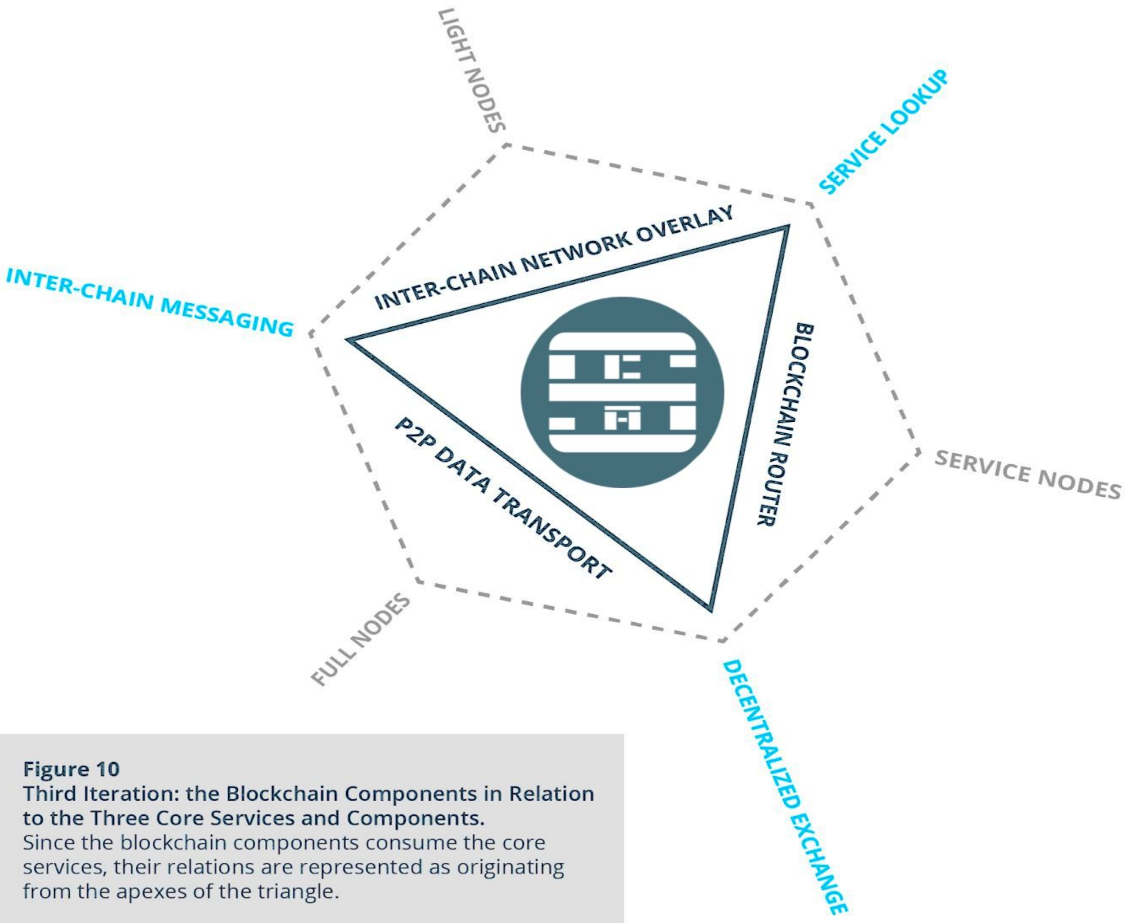


Figure 10
Third Iteration: the Blockchain Components in Relation to the Three Core Services and Components.
Since the blockchain components consume the core services, their relations are represented as originating from the apexes of the triangle.

BLOCKCHAIN SERVICES

Blockchain services supplement the general-purpose **core services** to support specific use cases. Since there is no limit to the number of inter-chain services that may be created on the Scalaris, there is no definable limit the number of blockchain services that third parties may build.

Scalaris DX, the first inter-chain dapp on the Scalaris, requires several blockchain services. These shall be introduced, both in order to document them, and to illustrate something of the nature of a blockchain service.

Overview of Scalaris DX Blockchain Services

To function as an exchange requires several services over and above an atomic swap between counterparties. In fact, any exchange (centralized or decentralized) must provide four key functions:

- capital storage
- order broadcast
- order matching
- settlement

Since Scalaris DX is a decentralized exchange, it follows necessarily that all four of these functions must be decentralized (in addition to some broader aspects pertinent to decentralization, like maintaining a fully open source codebase and virtually anyone being able to list a coin). With atomic swaps, the **decentralized exchange service** decentralizes both capital storage and settlement *a priori*. However, atomic swaps do not amount to an exchange on their own; order broadcast and matching must be decentralized in addition. It is these that are supported by blockchain services:

- Order broadcast is decentralized by leveraging the **service lookup** service, which is entirely peer-to-peer throughout.
- Order matching is performed locally by each trader's dapp, which are orchestrations of the **core components** and one or more blockchain components.

The principal design consideration for order broadcast and matching is that, like other peer-to-peer systems, they are naturally vulnerable to DOS (denial-of-service) attacks. In an analogous manner to how Bitcoin functions as an electronic cash system that solves the Byzantine Generals' problem, a workable solution must amount to a quality-of-service guarantee about orders on the book. Secondly, one of the design considerations in the token ecosystem is how to **monetise a service**, since in a decentralized ecosystem, if business models are not cryptoeconomically sound, they are not sound at all. What follows is an introductory discussion that frames these problems, and the Scalaris solution to them.

Decentralizing An Order Book

Note: the following sections detail the Scalaris candidate order system. While other systems are under consideration, the current one is included in order to introduce the design space in which decentralized order systems exist, and to encourage comment and contribution in this new area.

An order book is analogous to a notice board upon which traders may add liquidity by posting bids or asks, and from which other traders may take liquidity by consuming bids or asks. Along with the acts of posting orders to the book or taking them come commitments to settle accounts once a match is made. Now in a decentralized context, an order book becomes essentially a public notice board, upon which anyone is free to post an order, where no central party controls people posting, and whose intentions need not be to trade non-maliciously.

As such, a decentralized order book requires (a) a means of ensuring that only good orders are posted to it (that is, it must prevent order spam), and (b) that if an order is matched, the counterparties are compelled to honour their commitments to settle (that is, it must prevent order-DOS). Now, surprisingly, blockchain is *not* a good technology to use for an order book, for two reasons: firstly, order books need to operate extremely quickly – at the very least, supporting a realtime user experience – but blockchains require a transaction (generally put, a truth-claim) to be at least several blocks deep in the chain before its truth is sufficiently established. Secondly, since blockchains require mining or staking to establish truth, this opens an opportunity for miners to gain privileged access to order information, and potentially to frontrun, favouring the matching of their orders if they mine the next block. As a result, a different system is required to decentralize orders.

A final novel fact about an order book is that, because orders are commitments to a *future* settlement act, no funds are sent or risked throughout the order broadcast and matching process, so parties using an order system are exposed to a far lesser degree of risk than that of payments or settlement. This offers an advantage in the design of an order system, as it is not necessary to accept the crippling performance penalty that the use of a blockchain would impart; the system may function successfully while tolerating a certain degree of untruthfulness. As such, the design requirement is, as a *minimum standard*, to prevent illicit activity from scaling beyond isolated acts. Specifically, the Scalaris solution:

1. Supports UTXO verification, that is, that coins offered in an order are spendable.
2. Renders order spam unscalable, specifically via a hashcash-style trade fee that has minimal impact upon honest traders but makes spam significantly costly.

3. Renders order-DOS unscalable, in the same manner as in (b).
4. Renders Service Node collusion economically infeasible (see **special properties of Service Nodes**).
5. Supports partial order matching, enabling several counterparties to consume an order without the abandonment of one trade locking the entire order until the refund transaction becomes spendable. This can supplement a trading strategy that avoids the opportunity cost of a counterparty abandoning a single high value trade by spreading capital over several orders across a price range.

What Do Decentralized Orders Look Like? - Self-Sovereignty

We hold that decentralization, in the context of orders, amounts to preserving *self-sovereignty* between actors on the decentralized exchange. Decentralization is not a well-understood term, and is frequently conflated with "distribution" and muddled up with notions pertaining to reaching consensus. Decentralization, as Vitalik Buterin explains clearly, is fundamentally about *control:*a decentralized system places no party under the control of another, except where the matter under concern is a shared resource, in which case all parties participate as equals. Distribution, by contrast, is simply spreading a task or role between several parties, regardless of whether any one party controls this. Finally, consensus between parties strictly pertains to the reaching of agreement about a fact or action, not to whether work is distributed or control decentralized. This does not imply, though, that to decentralize Bitcoin does not require the reaching of consensus about which transactions are legitimate, and that work is not distributed over the edges of its network. These factors are necessary but not sufficient to the decentralization of Bitcoin.

Decentralizing a currency is analogous to decentralizing orders where *self-sovereignty* is encountered in the former: in the same way that any holder of bitcoins may send coins without third-party control, any trader may add liquidity to the order book or take liquidity from it; in the same way that Bitcoin users may prove to themselves that a given coin was sent at a given time, counterparties may verify the validity of orders and order-acceptance messages on their own. However, the analogy with cryptocurrency ends upon consideration of the fact that orders are *offers* to send coins, which require a future counterparty to consume them, and involves a matching process. Because a match requires the parties involved to mutually sign an order-acceptance message, and because *no other entities* need be required to establish that an order is matched, peers on a network may determine and discover the state of orders on a purely self-sovereign basis, without requiring a consensus algorithm (e.g. proof-of-work).

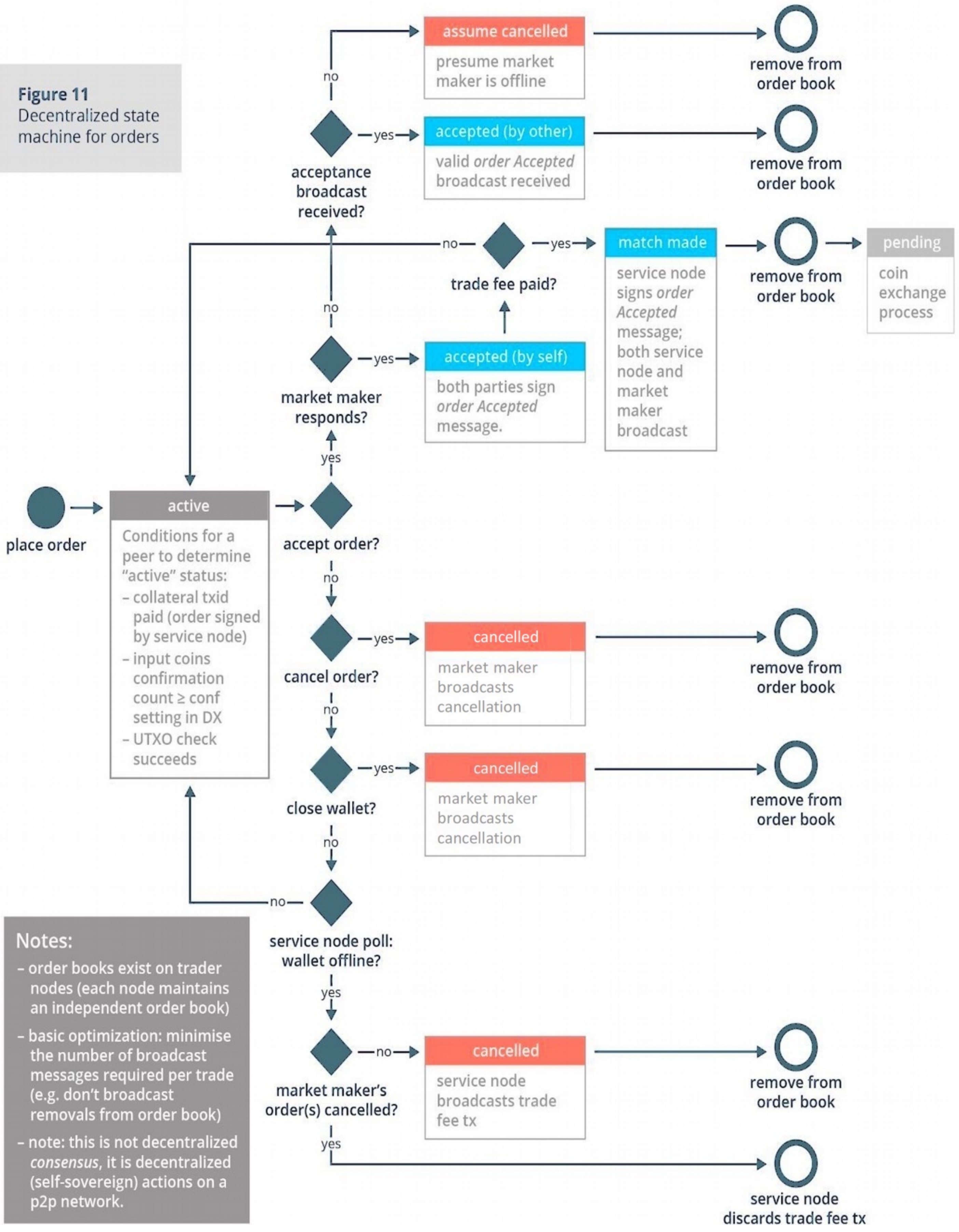
We propose an entirely self-sovereign order system design, involving three parties. Below is a cursory overview, which shall be developed upon as the discussion progresses:

- Before an order is broadcast, a market maker privately sends the order and an anti-spam fee transaction to a Service Node, which can broadcast the latter, spending the coins as a network fee and costing the maker money if it act maliciously. Upon validation of the fee transaction, the Service Node signs the order, which would-be market takers use to validate the order.
- When an order is broadcast, traders may verify, on their own, that (a) the order is backed by real coins, and (b) that it has been signed by a Service Node (signifying that a trade fee has been paid).
- When one or more traders attempt to accept an order, the market-maker adjudicates between these requests (standardly accepting the first valid request) and, on its own, selects its counterparty.
- When a counterparty is selected, it is in the maker's interest that the counterparty will not DOS the trade, and so it awaits verification by a Service Node that a trade fee was paid.
- Once a Service Node broadcasts a signed acceptance-message, the rest of the market updates its order books by removing the order from the book.

As such, Scalaris DX's order book functions as a decentralized state machine. The following diagram represents the state machine at a high level (the sections that follows introduce certain details in the diagram):

Orders: Decentralized State Machine

Figure 11
Decentralized state machine for orders


place order
active
Conditions for a peer to determine "active" status:
– collateral txid paid (order signed by service node)
– input coins confirmation count ≥ conf setting in DX
– UTXO check succeeds
accept order?
yes
no
market maker responds?
yes
no
acceptance broadcast received?
yes
no
assume cancelled
presume market maker is offline
remove from order book
accepted (by other)
valid *order Accepted* broadcast received
remove from order book
accepted (by self)
both parties sign *order Accepted* message.
trade fee paid?
yes
no
match made
service node signs *order Accepted* message; both service node and market maker broadcast
remove from order book
pending
coin exchange process
cancel order?
yes
no
cancelled
market maker broadcasts cancellation
remove from order book
close wallet?
yes
no
cancelled
market maker broadcasts cancellation
remove from order book
service node poll: wallet offline?
no
yes
market maker's order(s) cancelled?
no
yes
cancelled
service node broadcasts trade fee tx
remove from order book
service node discards trade fee tx

Notes:

- order books exist on trader nodes (each node maintains an independent order book)
- basic optimization: minimise the number of broadcast messages required per trade (e.g. don't broadcast removals from order book)
- note: this is not decentralized *consensus*, it is decentralized (self-sovereign) actions on a p2p network.

Separation Of Roles: Service Nodes

The Scalaris order system is a three-party system, rather than simply being between the two counterparties to a trade, and this is perhaps surprising. However, it comes as the result of considering the impact to the users experience of precisely when a fee is charged. For example, it would be possible to implement a simple decentralized fee solution if users were required to pay a trade fee prior to placing an order, yet this would result in their being charged even if they were to cancel an order, and even if their counterparty were to abandon a trade. In order to obtain the desired behaviour (that is, for users to only be charged a trade fee if the trade completes or if they cancel), yet simultaneously for fees to be paid upfront, prior to placing an order, a three-party system is implemented. It is the result of separating the incentives to offer liquidity, to take liquidity, to check that one's counterparty has paid a fee, and to earn trade fees without (a) being tempted to broadcast a trade fee transaction maliciously while (b) not charging trade fees unless an order is accepted. Again, the overriding intention is self-sovereignty, and this is preserved by separating and aligning each party's incentive to behave in particular ways. This is a complex system, and for simplicity only a few considerations shall be discussed at a time; at this stage in the discussion, the point is merely that Service Nodes are required to charge trade fees without requiring them to be charged when a party cancels before an order is accepted (see the **detailed protocol sketch** above for the exact roles and sequence in the order system).

Special Properties Of Service Nodes

To support a reliable mechanism for accepting but not spending trade fees (anti-spam and anti-DOS fees) if the party cancels the order before it is accepted, Service Nodes have several novel properties.

1. ### Limited in Number and Easily Identifiable

   Because trading counterparties rely on trade fees as an anti-spam and anti-DOS measure, it is necessary for them to easily identify whether a fee was paid, and this requires to the ability to differentiate orders (and order-acceptance messages) signed by a Service Node from other nodes' signatures.

   To achieve this, we require Service Nodes to hold 2500 SCA. That way, any trader may easily scan the blockchain for 2500 block UTXOS and compile a list of Service Nodes. Then, if one address containing 2500 SCA validates an order's signature, it has been signed by a Service Node.

2. ### Expensive to Act Maliciously

   Service Nodes shall be required to hold SCA for at least 1000 blocks before they may begin earning trade fees and block rewards. If a Service Node acts maliciously – that is, by broadcasting a trade fee when a user cancels an order legitimately – then the trader involved may submit a blacklist proof-claim to the network about the Service Node. To escape the blacklist, the SCA would need to be moved to a new address, in which case the Service Node would lose out on very many trade fees and block rewards while waiting 1000 blocks.

3. ### No Profit from Acting Maliciously

   In addition to the high opportunity cost of acting maliciously, a Service Node cannot directly or predictably profit from illicitly spending a trade fee, because trade fees are awarded to the next winning staker, which occurs probabilistically and is thus indeterminate with respect to the winner. As such, Service Nodes have no practical profit motive to act maliciously.

4. ### Supportive of Cryptocurrencies and Tokens Interoperable with the Scalaris

   To validate an order or order acceptance message, Service Nodes are required to verify that the address(es) in the order contain enough coins to fund the order. As such, they have significant hardware requirements, as they are required to maintain full node wallets of each blockchain interoperable with the Scalaris. This requirement further equips Service Nodes to relay messages and transactions for SPV and light nodes where necessary.

5. No Opportunity to Practice Insider Trading

   Because any Service Node may sign orders, it is not practicable for a Service Node owner to privilege the orders of any one trader (for example, her own trading node), because if it delays signing the orders (or order-acceptance messages) of others, they may trivially obtain good service from any other Service Node.

As a result of the above properties, traders have (a) a “good reason” to trust the testimony of a Service Node, (b) the ability to inflict income-loss upon a Service Node in the event that it does not act truthfully, of an order of magnitude greater than the loss of a trading fee. (c) As observed above, the low level of risk that traders are exposed to during an order process, and the severe performance penalty that would be imposed to obtain a higher degree of certitude, does not warrant their obtaining a higher degree of certitude about orders.

Order System Protocol

Assuming the preceding discussion of the general nature and realities of a decentralized order book is correct, the following protocol sketch is its natural solution. Please note that this solution is an area of active research by contributors to the Scalaris and may be further refined.

1. Maker Prepares Trade Fee
   a. Maker wants to buy *x* LTC with *y* BTC
   b. Maker calculates tx fee (payable in SCA): $y * 0.05\% / \text{price}_{SCA}$
      i. Note: price_{SCA} is the average buy price for a market-buy of SCA at the time of posting the order.
      ii. Note: SCA should be bought in advance of placing an order (advisable for speed), e.g. upon starting up the DX app - but only when the app's SCA supply decreases below, say, 1 SCA. Step (a) above is simply to calculate the quantity of SCA to spend.
   c. Maker creates but does not broadcast $tx_{spamfee}$, a trade fee, charged to market makers, in order to prevent order spam:
      i. a SCA tx
      ii. with its network fee set to the correct (0.05%) trade fee,
      iii. and the output as the maker's address.

2. Maker Posts an Order
   a. Maker creates an order to buy *x* LTC with *y* BTC. Fields:
      i. Value of *y*
      ii. Value of *x*
      iii. BTC address(es) funding the order
      iv. LTC address to receive coins
      v. Maker's XChat contact details (address, pubkey)
      vi. [other useful data: expiration, etc.]
      vii. txid of $tx_{spamfee}$
   b. Maker sends both the order and $tx_{spamfee}$ to the Service Node over XChat
   c. Service Node validates the fee against the order:

        i. txid of $tx_{spamfee}$ is identical with the $tx_{spamfee}$ txid field of the order
        ii. $tx_{spamfee}$ network fee is 0.05% of *y*.
            1. Note: Service Node must check SCA's price on the DX to calculate the correct fee; to account for volatility, fee must be within 15% of the correct value.
        iii. $tx_{spamfee}$ SCA address holds sufficient SCA to cover fee.
        iv. No txs in the mempool spending the same SCA address.
    d. Service Node signs the order.
    e. Both the Service Node and the maker broadcast the order + signature.
    f. Service Node holds $tx_{spamfee}$ in memory and does not broadcast it immediately.

3. Taker Prepares Trade Fee

a. Taker wants to buy *y* BTC with *x* LTC
    i. Note: from a UX perspective, the taker will most likely place a market or limit order for different quantities; this scenario assumes that a **matching protocol** exists to pair specific quantities between counterparties and thus to partially fill orders.

b. Taker calculates tx fee (payable in SCA): *x* * 0.2% / price $_{SCA}$
    i. Note: price $_{SCA}$ is the average buy price for a market-buy of SCA at the time of accepting the order.
    ii. Note: SCA should be bought in advance of placing an order (advisable for speed), e.g. upon starting up the DX app - but only when the app's SCA supply decreases below, say, 1 SCA. Step (a) above is simply to calculate the quantity of SCA to spend.

c. Taker creates but does not broadcast tx_{DOSfee} :
    i. a SCA tx
    ii. with its network fee set to the correct (0.2%) trade fee,
    iii. and the output as the taker's address.

4. Taker Accepts an Order

a. Taker sends an order acceptance message to the maker over XChat. Fields:
    i. LTC address funding the order
    ii. BTC address to receive coins
    iii. txid of tx_{DOSfee}
    iv. Signed using privkey for BTC address

b. Maker validates the acceptance message:
    i. Validly signed by maker's privkey corresponding to maker's BTC address
    ii. BTC address contains sufficient balance
    iii. No txs in the mempool spending from the BTC address
        1. *Trust note: even though the maker can validate the acceptance message, it should not proceed until tx_{DOSfee} is also validated by the Service Node, because otherwise the taker may DOS the swap for free.*

c. Maker signs the order acceptance message

d. Service Node validates the order acceptance message:

    1. *Trust note: the maker must validate the acceptance message before the Service Node does, or else the Service Node will (a) obtain information about buy interest at a given price prior to a counterparty being found, and (b) may privilege validating any one acceptance message over another or (c) may block/filter acceptance messages using arbitrary criteria.*

    ii. txid of tx_{DOSfee} is identical with the tx_{DOSfee} txid field of the acceptance message

    iii. tx_{DOSfee} network fee is 0.2% of *x*.

        1. Note: Service Node must check SCA's price on the DX to calculate the correct fee; to account for volatility, fee must be within 15% of the correct value.

    iv. tx_{DOSfee} SCA address holds sufficient SCA to cover fee.

    v. No txs in the mempool spending the same SCA address.

e. Service Node signs the acceptance message

f. Both Service Node and taker broadcast acceptance message.

g. Each trader's order book parses the message, signed by both maker and Service Node, as an order state change from "open" to "filled," and removes the order from its book.

5. Proceed to Bail-In tx Setup

    a. As per the atomic exchange protocol.

Conditions to Broadcast Trade Fee Transactions

A Service Node shall not broadcast a trade fee transaction if:

- The order expires.
- The payer closes the app without anyone accepting the order.
- The order is cancelled prior to it being accepted (step 4f above).
- The counterparty fails to broadcast its bail-in transaction.

Order Matching System

The Scalaris requires a decentralized system for (a) translating standard order-types on exchanges (market, limit, etc) into basic liquidity-consumption events, and (b) for matching quantities of one coin with quantities of another coin at specified prices. Adopting the decentralized order state machine presented above – that is, where parties adopt self-sovereign roles as either maker or taker – order matching consists of takers requesting to consume liquidity from makers, and in the case of limit orders, the presence of absence of orders up to a certain threshold functioning to determine the trader's role as a maker or taker.

The behaviour of a market order is as follows: consume orders on the book, beginning from the best-priced order, moving to the next-best-priced, and so forth, until the market order is completely consumed. If the market order consumes all orders on the order book without being completely consumed, then cancel the remaining amount on the order.

The behaviour of a limit order is as follows: if the order is a sell and the price of the order is lower (or if it is a buy and the price of the order is higher) than the highest (lowest) counteroffer on the order book, then consume orders on the book until either the user's order becomes the lowest (highest) offer, or is completely consumed. If the former, then add the remainder of the trader's order to the order book.

To avoid issues with change on UTXO-based cryptocurrencies – and the resulting need to wait for the change to confirm before the rest of an order can be traded – XBridge shall automate the distribution of trading wallets' coins into small, regular amounts at separate addresses, in order to maximise the number of outputs that are completely consumed by a counterparty and minimise change to trivial levels. Additionally, a minimum trade size shall be imposed that corresponds to the size of the amounts per address, to prevent the malicious creation of change.

Trade History Service

A history of trades between coins is required for charting and other technical analysis tools, and in general for traders to obtain information about a market upon which to make trading decisions. The truthfulness of a trade history is of paramount importance, because considerable advantage over other traders may be gained if one were to fabricate it. As such, the trade history of any given currency pair shall be provided in a "trustless" manner as a Scalaris service.

The solution sketch that follows is somewhat idealised, for reasons of simplicity and in order to present the leading idea rather than the detailed technical solution; a production-ready solution would be considerably more succinct (mostly due to not including the trade history data itself in one of the the atomic swap transactions), and instead would employ a more sophisticated zero-knowledge proof scheme like, for example, bulletproofs.
A "trustless" dataset would require (a) a suitably truthful means of committing trade history data to it, and (b) an equally truthful means of retrieving data from it. The solution is as follows:

A trade history blockchain would be created, and its nodes' work in establishing consensus would amount to committing trade transaction data from other blockchains to the trade history blockchain. Typical data committed would include:

- coin A
- coin B
- quantity of coin A
- quantity of coin B
- price of coin A : coin B
- time that the first bail-in tx was spent

The data committed must be sufficient for consumers of the data to synthesise into many forms. For example for TradingView charts, the following data would be required:

- period of candle
- start time of candle
- open price
- close price
- high
- low
- and so forth

Commitment Proof Data

To commit trade data, nodes on the trade history blockchain must submit *proof data*:

- coin A
- coin B

- txid of spent bail-in transaction on chain A
- txid of spent bail-in transaction on chain B (or mark if absent)
- txid of spent trade fee transaction for market maker
- txid of spent trade fee transaction for market taker
- timestamps of all transactions
- timestamp for the transaction in which the proof data is submitted

Proof data submitted by a node is then verified by the network by searching the blockchains of the coins involved in a trade and validating the proof data against the the coins' blockchains. Nodes would also engage in a deduplication exercise similar to mining with respect to the fact that the network must determine the first node to submit proof data for a given trade, and discard proof data submitted by other nodes.

Data Retrieval Methods

There are multiple way to retrieve trade history data. Traders may either:

- download the trade history blockchain, and thus retrieve the data for free
- store in local memory all completed trades – which would require running Scalaris DX continually (in practice, this would only be useful for updating a chart in realtime)
- request trade history from a node on the trade history chain, for a fee:
  - for a specified coin pair
  - over a specified time

Supply of Trade History Data

To supply trade history to traders, nodes on the trade history blockchain must submit:

- a hash of all txids for coin A, coin B, and trade fees within the time range specified by the trader
- the node's address

With this data, the trader constructs a simple zero-knowledge proof (though see the note above) of the truthfulness of the data by requesting trade history from many nodes on the trade history blockchain, and verifying that the supplied hashes are identical. If the results are identical, then this amounts to a low probability that the data is untruthful, because the nodes have no good reason to trust one another and are thus not in a strong position to collude maliciously. If the trader wishes to obtain greater certainty about the truthfulness of the data (s)he may request trade history from mode nodes, or download the blockchain herself. Nodes may further monitor each others' responses to traders and punish dishonest nodes by submitting proofs of how their response is unfaithful to the record in the blockchain; on this basis it is trivial for the network to reach consensus about the dishonesty of a node and blacklist it.

If the trader is satisfied with the response(s) from the trade history nodes, it would pick the first node to supply the hashes, and commence an atomic swap with the following properties:

- a bail-in transaction only spendable using
  - a private key corresponding to the address the node provided, and
  - the trade history data corresponding to the hash provided
- (In other words, the trade history data functions as the secret in the atomic swap)

As such, if the trade history node spends the bail-in transaction, then it must reveal the trade history data, and so the trader receives it. Simultaneously, the trade history data cannot be revealed without the trader paying for it.

The current (and simplistic) sketch has a few properties worth noting:

- The size of the requested dataset may be limited by the maximum field length for the secret in the transaction format. This, then, drives revenue for trade history nodes by requiring traders to submit more than one request if they desire trade data over a longer timeframe.
- The fee per dataset may be dynamically adjusted against the trade volume, since greater volume per unit time would reduce the time span of a dataset of a certain maximum size.
- Mild obfuscation of trade history data is provided against other traders intercepting the requested trade history data once the trade history node reveals it: because they neither the trading pair nor the timeframe is included in the dataset, it will not be obviously useful to them and would be expensive and complex to synthesise.
- Strong obfuscation of trade history data would be afforded by it being sent encrypted over XChat, and the secret in the atomic swap functioning also as a decryption key for the data. This, however, would require a more sophisticated zero-knowledge proof that the one in this sketch (see above).

Registry Service

The trade history service above appears to be generalisable to provide a workable registry service for inter-chain services. Intuitively, where the trade is not for a coin but for a digital good instead, the commit and lookup phases in **blockchain routing** remain unchanged. What would be required in addition is for trade history nodes to filter their blockchain's record of trades for the most recent record in which a chainId appears, and to compile the resulting list of chainIds and their associated serviceIDs. This list would be delivered, instead of trade history, using the protocol in the preceding section.

Project Phases

The following section is a broad outline of the long term course of the project, intended for gauging overall scope, and is not a series of commitments to development milestones. Shorter-term roadmaps, with well-defined milestones, are published as necessary.

PRODUCTION MVP

- Monolithic client/node
  - Blockchain router
  - XChat protocol
  - Service monetisation mechanism
  - Trade fee distribution mechanism
- Decentralized exchange dapp
  - Frontend UI
  - Market, limit, and stop orders
  - Order book
  - Order history
  - User open orders
  - TradingView charts integration
    - Uses each user's account API credentials
  - Setup wizard: automatic wallet API and charting API configuration
  - Risk control (over number of confirmations acceptable)
    - Filters order book
    - Allows you to place orders sooner
    - Change handling:
      - Order accepted; change is returned
      - Change unspendable for x minutes
      - Goes into appropriate risk bracket
      - Order's risk bracket auto-updates with coin age

PHASE 2

- Modularise xbridgep2p
  - Blockchain router module
  - XChat module
  - Coin exchange module
  - Decentralized exchange client
- APIs for all modular components
- Support for data payload in exchange protocol

- Easy interoperability between exchange protocol and xchat transport protocol (controlled via your own Dapp)

PHASE 3

- Support for further order types: trailing stop, OCO
- Support for leaving orders in book after closing app (orders committed to blockchain)

PHASE 4

- Protocol enhancement: derivative market for swaps (p2p margin lending)
- Protocol enhancement: generic derivative markets

Technical Specification

For easier maintenance, and in order to retain a single source of truth for low-level documentation, this section has been moved to GitHub.

MESSAGE SEQUENCES

Forthcoming. See https://docs.scalaris.info/ .

API REFERENCE

See https://api.scalaris.info/.

Use-Cases

Infrastructure for a not-yet-existing ecosystem presents some difficulties to the imagination. “What is this for?” is the single most common design-focused question asked, and the correct answer is something like “anything that can benefit from a token ecosystem” – which is most things. For a less abstract answer, here is a short list of use-cases for the Scalaris:

1. Decentralized Exchange

The decentralized exchange of crypto-tokens is in fact a core service of the Scalaris, since it is necessary for the monetisation of any other service.

This, within an easy-to-use dapp UI, is also the Scalaris first consumer product, since it fulfils a real need in the crypto-community for decentralized trading technology.

The prevalence of hacks, fraud, failure, and theft from centralized crypto exchanges has resulted in an alarming 1 in 16 Bitcoins being stolen. A workable decentralized exchange will thus provide a vital enabling function in the fledgling token ecosystem for the safe, secure exchange of coins and tokens

2. Blockchain Router

Blockchain routing is also a core service of the Scalaris, since inter-chain traffic must be routable to its intended destinations. That said, it is also consumable as a valuable service, which any node may require in order to either deliver or consume an inter-chain service. The Scalaris incipient router, XBridge, currently delivers a free service, and this may remain so for the indefinite future.

3. Inter-Chain Messaging

Whether used as a chat app or as a data transport, inter-chain messaging is an indispensable service for the token ecosystem. As with decentralized exchange and blockchain routing, this is a core Scalaris service, and it goes by the name of “XChat.” It is end-to-end encrypted, peer-to-peer, and may be used for the ultra-secure delivery of digital goods and messages. It is currently a free service, and is (currently) packaged alongside the blockchain router in XBridge.

4. Mobile App Exploiting Multiple Chains

A mobile app, with its small footprint, would most likely have only one SPV node and its native blockchain token. As such,

- It would consume services, not other coins
- Various **blockchain services** it consumes shall run Scalaris components
- When the app requests a service, the service shall generate a “secret” which is also a decryption key for the digital goods
- The service shall send data enabling the app to construct a zero-knowledge proof that the goods are legitimate
- The service shall create a bail-in transaction in an **atomic swap**
- The service shall spend the bail-in transaction and later trade it for another coin, if preferred
- The app thus receives the secret and may consumes the service

5. Near-Perfect Coin Mixer

A private currency such a ZCash, ZCoin, or Monero may integrate to XBridge, and script the automatic trading of any currency for the private currency and back again into the original currency. Since the decentralized exchange does not require any third party to be trusted with users' data, and since the atomic swap involves no counterparty risk, the result is a nearly perfectly private mixing service.

6. Decentralized Marketplace Spp

A marketplace app would typically require the following services: (a) customer reputation and info, (b) payment processing, (c) image storage, (d) item listings. A microservices architecture is advisable for the **reasons given above**, gaining the advantages of utilising multiple blockchains. Thus, one chain may store encrypted customer info (see item 13 on this list), use the XBridge to accept payments in any cryptocurrency, store images on a server, and use a third chain and in-wallet code for the item listings and UI elements. The result is a scalable, composable set of services that are easier to bugfix, upgrade, or replace.

7. Fuel-Converter for Ethereum Smart Contracts

Using the decentralized exchange, any Ethereum contract may be supplied with "gas" in other coins.

8. Truly Decentralized Stablecoin

A stablecoin may maintain its peg by exploiting the fact that trade records on a decentralized exchange are on-chain. As such, a provably truthful dataset is available for determining whether to mint or burn coins (or freeze and unfreeze them) in order to maintain a peg.

9. Self-Sovereign ID and Personal Information Manager

A personal information service may records encrypted personal metadata on a given blockchain equipped with a revocable permissioning system. Users thus acquire self-sovereignty over their personal information. From this point, one may integrate this blockchain to any website or app requiring sign-in, or users can voluntarily sell their metadata to advertisers for micropayments, or it can support passport/identity systems. Emerging technologies positioned to exploit this use case are Bitnation and Microsoft's Coco Framework.

10. Supply Chain 2.0 Solution

Scalaris infrastructure is well-suited to function as a "supply chain 2.0" backbone. Parties typically find themselves on different blockchains, with a need to interoperate, and may do so by exploiting Scalaris services. Multichain apps will thus be able to read data from several chains, whether they are specialized in shipping data like Bill of Lading, product manufacturing data like Bill of Material, financial dat, and so forth. By comparing metadata from several sources, the Scalaris could empower companies to limit attack vectors like invoice spoofing and certification counterfeiting.

11. IoT Infrastructure

Some perennial IoT security issues may be solved by exploiting blockchain technology, and then interoperating between thousands of blockchains over the Scalaris. Diverse opportunities for granular monetisation present themselves too: for example, one may accomplish transaction batching on several chains at once using SPV wallets. The data stream may thus be tokenized, and nodes may be incentivised to engage in pattern finding in a company's big data.

12. In-App Ad Service

A mobile app may earns its users tokens by screening ads delivered to the app as a Scalaris service. Tokens could then be used to power inter-chain service consumption for the app, delivering a "free" service for the user, but a monetised one for the service providers.

13. Decentralized p2p Storage Solution

A blockchain-based storage solution, such a Storj, may have its user base significantly enlarged and monetized via inter-chain service delivery.

14. Permissionless ICO Platform

Anyone may offer token sale over a decentralized exchange, with no permission required.

15. Business Case Tool for Distributed Budget Management

Crypto projects are typically launched as a big crowdsourced business case (ICO), where a budget is negotiated with the market. However, the actual account balance fluctuates as the price of the cryptocurrency the crowdsourcing was conducted in. changes in value. Using the Scalaris, developers may manage the distribution of tokens and accounts across chains. Furthermore, with the use of a smart contract, disbursements and investments in other coins may be managed and, in general, the project's business plan coded and automatically executed by the contract with perfect transparency.

16. Integration to ERP, CRM, PLM Systems Across Companies

The Scalaris simple API-based integration enables interoperability either directly or indirectly with consortium-type and private blockchains, like those of ORACLE and SAP.

17. Infrastructure for the Internet of Value

The Scalaris inter-chain infrastructure shall function increasingly over time to create an "internet of value" that is inherently truthful, transparent, and fairly available. As companies' general- and sub-ledgers come gradually to interface with other companies' ledgers through blockchains, the resulting network of blockchains will become a full representation of value streams and of a given system's value. This enables advanced and deep awareness of value across entire systems, with correspondingly powerful and far-reaching consequences for the financial system.