Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Towards Datamarkets with Bitcoin

Master Thesis

Francisc Nicolae Bungiu

fbungiu@student@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Christian Decker, Dominic Wörner, Laura Peer
Prof. Dr. Roger Wattenhofer

September 22, 2015

Abstract

In recent years, there has been a widespread expansion of data collection and complex methods to analyze such collected data. Everyone is constantly generating data by using a large range of computers, smartphones, and gadgets. In addition, there is an emerging trend towards the Internet of Things technologies that consist of billions of sensor nodes bridging the gap between the physical and the digital world, and creating massive amounts of data, but with no incentive to share.

In order to provide an incentive for the sensor node owners to share the generated data, these sensor networks have to initiate data markets that interested customers can subscribe to and pay for the acquired data. Bitcoin provides an Internet-native payment mechanism and protocols on top of Bitcoin are able to support small payments and avoid high cumulated transaction processing costs.

This thesis proposes a centralized secure scheme that allows data purchasing from any Internet-connected sensor node using Bitcoin payments. Based on micropayment channels to aggregate payments and minimize transaction fees, and on contracts between the protocol participants to minimize trust, the scheme allows human judgement to be taken out of the loop and supports complete automation.

Contents

Chapter 1

Introduction

The Internet of Things (IoT) is a novel concept that has rapidly expanded in the last few years, refering to the network formed by any physical object, embedded with sensors and connectivity —such as Radio-Frequency Identification (RFID) tags, sensors, smartphones, etc. —that enables such nodes to exchange data. This technology allows these objects to be sensed and controlled remotely ([4]), having a great impact on the every-day life of users, and resulting in efficiency and financial benefits.

Sensor nodes with Internet connectivity enable attractive sensing applications in various domains, ranging from health-care, security and surveillance, event forecasts, environmental monitoring, agriculture automation, energy consumption, transportation, etc., creating data markets in which they can actively participate. However, IoT is still lacking an efficient payment method and is limited by high transaction costs. [16] introduces the concept of Sensing as a Service, in which a number of sensors offer their generated data as a service to interested entities through a central operator for an agreed cost.

Most existing payment protocols for the Sensing as a Service model involve a third-party to process payments and arbitrate disputes, which results in higher costs and reduced efficiency. Another approach is to rely on Bitcoin, a decentralized electronic payment system. Bitcoin has at its root the block chain, a transaction database that is shared by all network nodes. However, the solutions following this approach fail to address the scalability problem of Bitcoin: they rely on atomic operations that are completed directly on the block chain, thus increasing its size. At the moment, Bitcoin supports less than 7 transactions per second with a 1 megabyte block limit, with a fee per transaction of 0.0001 BTC, equivalent to 0.02 USD. Broadcasting a high volume of small-value transactions for each individual data query is clearly not feasible since Bitcoin does not support such a high transaction rate, and the associated fees could easily exceed the transfered amount.

This project proposes a centralized, low-trust and secure scheme that allows and incentivizes sensor nodes to share sensed data and get paid in exchange in bitcoins. The presented solution utilizes micropayment channels to solve Bit-

coin's scalability problem by bundling several small payments for each set of acquired data and only publishing the final, aggregated transaction to the network. In order to confer a low-trust relation between the system entities, the protocol relies on contracts that pass the payment obligation down from the buyer to the central coordinator, and then to the sensor node. In addition to the theoretical model, a proof-of-concept is built based on the Android built-in sensors.

1.1 Background

In the following sub-sections, the background on the used concepts and sub-protocols is presented.

1.1.1 Bitcoin

Since its invention in 2008 [10], Bitcoin has grown into a global electronic payment system. It uses peer-to-peer technology to transfer funds with low transaction fees and no need for a central authority such as a financial institution, making it an attractive alternative to traditional payment methods. A proof-of-work system and cryptographic protocols are the building blocks of the protocol, with peers participating in the network being responsible for processing transactions and issuing of bitcoins.

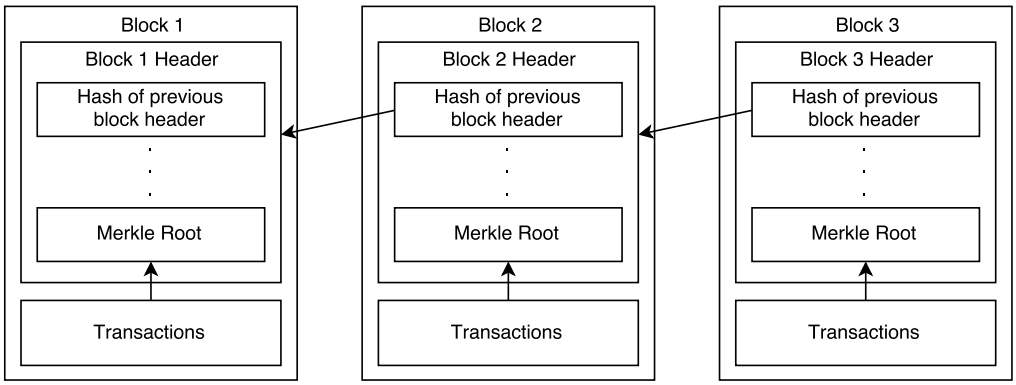


Figure 1.1: A simplified representation of the Bitcoin block chain.

Bitcoin relies on public key cryptography to authenticate transactions. Each peer in the network has one or more public and private key pairs that are stored in a file called a wallet. In order to support and authenticate transactions, an address is derived from the public key and disclosed to the participants. Transactions consist of one or more inputs (references to previous transaction outputs) assigning the value of all inputs to one or more outputs. An output

consists of a tuple of a specified bitcoin amount, and an output script, called scriptPub. When trying to claim the funds, an entity must create a signature script (scripSig), which has to satisfy the conditions in the previous output's pubkey scripts. In most cases, the script only requires a signature matching the address, which proves the possession of the corresponding private key. The difference between the total input amount and the total output amount in a transaction represents the transaction fee. A peer holding a private key can sign a transaction spending some of its wallet's amount to some other address, and any other peer that sees this transaction can verify the signature using the peer's public key. In order to claim an output, the payee must prove that it possesses the private key corresponding to the public key included as the destination of the transaction. Thus, the available funds are represented by the amount of all unspent transaction outputs the peer possesses a private key for.

In order to be validated and accepted by the peers, a transaction needs to be broadcast to the network. Special network nodes, called miners, will then timestamp it by applying a hash function into a continuously growing chain of hash-based proof-of-work. This forms a record, called the block chain, containing all transactions in Bitcoin ever made, that cannot be changed without redoing the successive proof-of-work. The public ledger is distributed to all network peers and is crucial for protection against double spending and modification of older transactions. The proof-of-work performed in Bitcoin is based on HashCash [5], which means new blocks containing one or more transactions are only accepted into the ledger if the hash of the block header contains a specified number of zeros as a prefix, adapted dynamically to produce blocks at a constant rate. This means that the expected time for a transaction to be included in a block, and thus, confirmed, is 10 minutes. The computation, called mining, is incentivized through fees that are earned by the nodes performing it.

1.1.2 Micropayment channels

Unlike traditional payment methods, Bitcoin transactions are very cheap in terms of fees, but still have a considerable cost given the mining and storing they require. The context of Internet of Things requires a high volume of small-value payments, thus the overall fees might reach and surpass the value of the actual transactions. In addition to this, broadcasting several transactions to the Bitcoin network in a very short time window will trigger network anti-flooding algorithms, which will result in either the transactions being delayed or, even worse, not relayed. Last, the payee of such small transactions will end up with a wallet full of "dust", small outputs that cost more to be spent than their actual value.

To address these issues, the construct of payment channels was introduced in Bitcoin [8]. After an initial setup process between two participants, the payer

can start sending tiny payments to the other party off the block chain at high speed, without paying high transaction fees, in a trust-free manner.

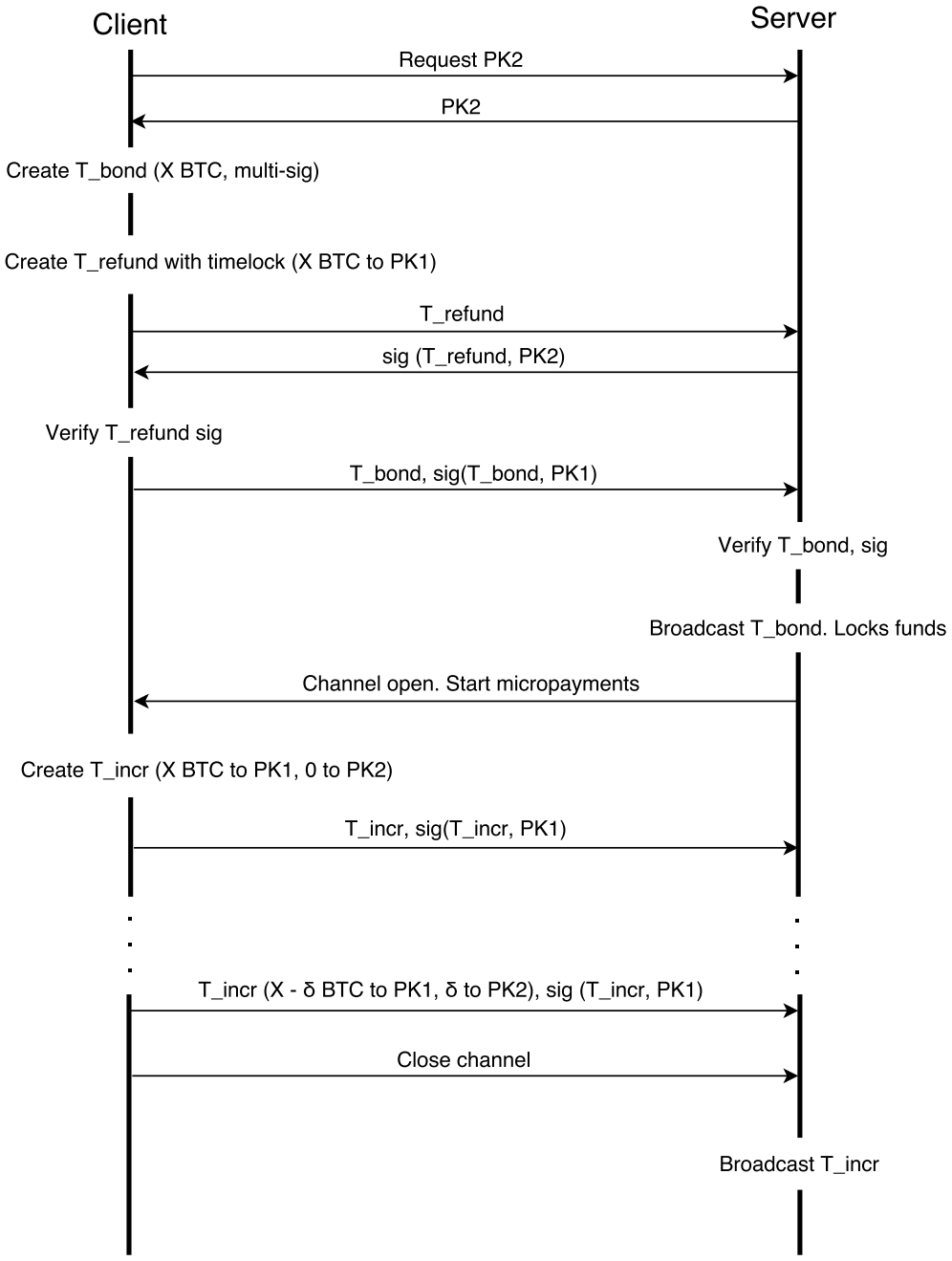


Figure 1.2: Micropayment channel protocol: setting up the shared account (bond transaction), creating the refund transaction, and updating the incremental payment transaction.

Consider a client that is interested in a certain service offered by a server. To benefit from the service, it will pay the server in bitcoins. Since none of the participants know each other, a zero-trust solution is necessary to reassure both client that it will not lose its bitcoins in case the server does not provide the data, and that the server will not provide the promised service without getting remunerated. The construct works in two stages. First, the client creates a multi-signature transaction, a shared account requiring signatures of both participants to spend from it, that pre-allocates a certain amount of bitcoins for use in the channel. This shared account is called the *bond transaction*. If the client signs and immediately sends this transaction to the server, the server could simply broadcast it and keep the money hostage. Thus, the client keeps the bond transaction private for now and creates another one, called a *refund transaction*, and sends it to the server to sign it. This transaction refunds the entire value to the client, but it *time locked* using the nLockTime feature of Bitcoin, ensuring it will not become valid until some time in the future (channel lifetime). If the server vanishes at any point, the client can then use the refund transaction to get all the money back at channel expiration time.

Once the refund has been signed by the server, the client can safely send the signed bond transaction. After verifying the signature, the server broadcasts it to the network, locking in the money and opening the channel between the two parties.

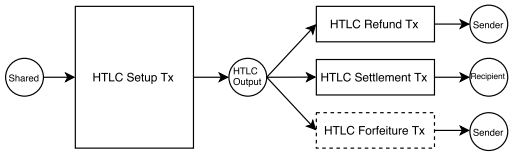
At this point, the work-and-pay cycles can begin. Initially, the client creates a new transaction spending from the shared account and adds two outputs: one to its own address with the full amount, and one to the address of the server. After signing it, the client sends it to the server. When a new service is requested, the transaction is updated by the client by increasing the value to the server and decreasing the value allocated to its own address. In each update cycle, the transaction is signed and handed over to the server. When the time comes and the client wishes to close the channel, it notifies the server, which in turn signs the transaction with the highest value alloted to it and broadcasts it to the network. This step closes the channel, unlocking the client's remaining money and making the server's money available for spending. The client could be tempted to broadcast an older transaction that gives less money to the sensor node than it deserved for services it benefited from, but it is missing the server's signature. Thus, the construct of micropayment channels prevents misbehavior of both parties.

1.1.3 Hashed Time-Lock Contracts (HTLCs)

Hashed Time-Lock Contracts, or *HTLCs*, are block chain enforced contracts that require the recipient of a payment to reveal a previously generated secret in order to claim the sent funds. This construct can be used to provide end-to-end

```
OP_IF
    2 <pubKey_A> <pubKey_B> 2
    OP_CHECKMULTISIG
OP_ELSE
    OP_SHA256 <H>
    OP_EQUALVERIFY
    2 <pubKey_A'> <pubkey_B> 2
    OP_CHECKMULTISIG
OP_ENDIF
```

(a) HTLC pubScript.



(b) HTLC Transaction structure.

security in multi-hop payments, by preventing the intermediaries from delaying or keeping funds for themselves.

To create an HTLC, a special *HTLC setup transaction* is created. Its output can only be claimed by the final recipient B of the payment using a previously created secret. In order to receive a payment, B has to generate a secret R and hash it to obtain H. Then H and the B's address are directly transferred to the payer A. At this step, all nodes on the payment path between the sender A and the recipient B need to create HTLC setup transactions connected to the output of a shared account using the provided hash H. The output of the HTLC setup contains a pubScript as shown in 1.3a, which requires a signature from both participants (first branch), or the next hop provides R' such that it hashes to H. Once all setup transactions are in place, the recipient B can release R to claim the funds from the previous node. The secret is then revealed step by step by all nodes on path all the way back to A, which completes the transfer.

To prevent the intermediaries from delaying the funds, stealing or keeping them hostage, HTLC refund, settlement and sometimes forfeiture transactions are created, all three types claiming the HTLC output. The *HTLC refund transaction* is similar to the refund that is created in the micropayment channel setup protocol, but has a higher time lock to give the sender enough time to react, should the receiver not cooperate. The *HTLC settlement transaction* guarantees the receiver it can pull the funds if it is in the posession of the secret. It is important for the sender to use different signing keys A and A' for the two branches of the HTLC output. Otherwise, the receiver could simply reuse the signature of the settlement transaction in the if-branch and claim the funds without ever revealing the secret, since the signature is valid for the entire pubScript. Lastly, the *HTLC forfeiture transaction* addresses the case in which the sender and the receiver agreed to void the HTLC and remove the output. If the receiver comes in possession of the secret at a later stage, it still has the signed settlement transaction that it could simply broadcast and steal the bitcoins. Because of this, every time the receiver backs off from the HTLC, it creates and hands over a forfeiture transaction that transfers the bitcoins back to the sender. Should the receiver broadcast an older setup transaction, the sender can simply use the

forfeiture to get the money back.

The main use of the HTLCs is to keep as many transactions as possible off the block chain. In order for this to work, the recipient, instead of broadcasting the settlement transaction to reveal the secret and claim the money, it can simply reveal it directly to the sender. This way, both parties can agree on removing the HTLC output from the setup transaction, updating the output allocated to the recipient with the amount of the HTLC. If the recipient chooses to void the HTLC instead, it can sign a forfeiture transaction and hand it over to the sender, guaranteeing the sender that it will not use an older setup transaction. By doing this, the HTLC output is again removed, and its value is added back to the output of the sender.

1.2 Related work

Following the expansion trend of the Internet of Things technologies, there has been an increasing interest in creating secure payment protocols that allow customers to acquire sensor data efficiently.

[20] introduces a new low-trust E-business architecture that is tailored for IoT. This architecture is based on Bitcoin, eliminating the need for a third-party in the process. In order to make a payment and receive the data in exchange, the buyer and the data provider make an offer-proposal exchange, then build a transaction that spends the required amount and publish it to the network. A similar approach is proposed in [11], where the encrypted data is directly included in the block chain after a payment is made, which only the buyer in possession of the corresponding private key can decrypt. In both approaches, all transactions, no matter how small, need to be published to the network, which results in high cumulated fees and time-wise inefficiency, thus being directly affected by the scalability problem of Bitcoin.

Work has also been done on micropayment protocols that reduce fee costs. Amazon Flexible Payment Service uses a central provider that aggregates clients' micropayments into macropayments that are flushed to the seller at specific times. This solution does not provide anonymity, since the central provider can keep track of all payments, and moreover, was discontinued on June 1, 2015. Other protocols ([9]) involve a probabilistic approach. Using a selection rate s, it discards all unselected micropayments and selects one with probability s that can be deposited for an amount $1/s$ times bigger than the original amount. This should ensure everyone gets, on average, the expected amount. However, this solution lacks anonymity, requires PKI certificates and comes closer to a bet than an actual transaction. The same probabilistic approach is adopted in [12], but this solution needs a third-party mediator if users are dishonest.

The [19] paper proposes an anonymous, semi-offline micropayment system for

transactions in IoT. In this approach, an electronic coin is obtained by iteratively applying a hash function to some initial seed, thus allowing the user to spend it in fractions, by presenting a set of hash chain nodes to the vendor. The protocol is designed to meet the needs of IoT transactions, however, it has limited potential to be implemented because it does not rely on an existing financial institution. Another coupon-based system has been proposed by Rivest in [15] and was improved by Payeras-Capella ([13]) to integrate anonymity and prevent users from exceeding their account limit. The drawback of the latter is that the financial institution needs to be contacted directly before each payment, and any unspent value is lost in favor of the financial institution. Wilusz [18] solves these issues by requiring a single contact with the financial institution at coin issue time and allowing the user to use unspent fractions in future transactions. However, a clearing house is necessary to lock in the coin by the vendor, which increases transactions costs, and poses the risk of locking the coin indefinitely.

It is worth mentioning that there have been several proposals and ideas that can be exploited to support trustless, instant, off-the-chain Bitcoin payments on the Bitcoin discussion forum [1] by Mike Hearn, Alex Akselrod, hashcoin, Meni Rosenfeld, C.J. Plooy, Tier Nolan etc.

On the Contracts page of the Bitcoin Wiki [8], Mike Hearn describes an idea on how to trade with somebody with no trust involved, based on multi-signature transactions. The funds get locked in a shared account controlled by at least two of the three participating parties: the client, merchant and mediator. If the transaction is successful or a refund is agreed upon, the client and the merchant can move funds. If the trade fails, the client and the mediator can agree, and a charge-back occurs. Finally, if the goods are delivered but the client does not want to fulfill its part of the agreement, the mediator and the merchant agree and the merchant gets the client's money. Mike also proposed the idea of trading across multiple currencies without a third party, and Tier Nolan formalized it in a protocol that allows the exchange atomically, using time locks and hash commits.

C.J. Plooy presented a draft that combines Bitcoin and Ripple to create a high-speed, scalable, anonymous, decentralized, low-trust payment network [6]. In this Bitcoin-specialized variation of the Ripple system, neighboring pairs have a shared account that is partly allocated to one of them, and the rest to the other, as agreed between the two, which lowers the needed trust. Using sequence numbers to update this transaction as payments in the network are made and lock times to prevent blocking the money indefinitely.

Meni Rosenfeld proposed a simple way of reusing existing micropayment channels instead of establishing new ones if a path between the payer and the payee already exists. The intermediaries would then only be trusted with the amount of a single payment from a single buyer.

Alex Akselrod proposed [2] [3] and built a proof-of-concept for chained micro-

payment channels with two-phase commits. The original micropayment channel payment scheme was modified to allow sending funds both ways using commit hashes and cycles of refund transaction updates with decreasing time locks. However, the proposal suffers from the Bitcoin malleability issue [7] and the possibility of getting money locked in indefinitely, should the contract transaction be broadcast before refund transactions are fully signed.

Peter Todd's proposal in [17] provides the an efficient way for establishing micropayment channels between peers with a central hub. This central entity plays the role of a router, forwarding payments from the payer to the payee. Suppose Alice and Bob have previously established micropayment channels with the hub. If Alice wishes to send bitcoins to Bob, she should normally establish a new micropayment channel with him. However, both of them have channels opened with the hub, so Alice can send the funds to the hub, and the hub can then forward the payment to Bob. Using Hub-and-Spoke payments, payments between buyers and service providers with a central coordinator can be made efficiently, saving costs in terms of fees. However, the forwarding process should be enforced and secured, part that is noted in the proposal, but without specifying an actual way to do so.

The Lightning paper [14] presents a promising solution to Bitcoin's scalability problem and provides a way to securely forward payments on a path of peers through block chain enforced contracts. It solves the issue by using timelocks on a network of micropayment channels combined with Hashed Timelock Contracts (HTLC) —recipient generates random data R, hashes it to produce H and sends H to the sender of funds, together with its bitcoin address. Sender routes its payment to the receiver and when an updated transaction is received, the recipient may elect to redeem the transaction by disclosing the random data R, which pulls the funds from the sender. With the introduction of a new sighash type which solves malleability, the proposed protocol allows offchain transactions between untrusted parties with fully enforceable contracts, making Bitcoin scale to billions of users.

Chapter 2

Design

2.1 Requirements

For the presented use case, a scheme that enables sensor nodes in an Internet-of-Things network to share sensed data and get paid in exchange must meet the following requirements:

- Provide an entity that the node can register sensor type availability to.
- Provide an entity that buyers can query for available data and that can provide sensor node contact information.
- Scalability: avoid broadcasting all micro-transactions to the block chain.
- Efficiency and speed: fast communication between components, payments and data should be confirmed and received instantly
- End-to-end security of the payments.
- Anonymous: linking of buyers and sellers and identification across sessions should not be possible.

2.2 System Overview

The design of the payment system achieving the requirements above comprises three major components: sensor nodes, a central hub, and data buyers. The first component consists of several sensor data providers :IoT sensor nodes, Android phones running specialized app, etc. The central hub plays an essential role in sensor node discovery, payment and data relaying. Last, the buyers' component consists of all entities interested in the sensed data. The communication between the buyers and the hub, and between the hub and the sensor nodes is done through TCP links.

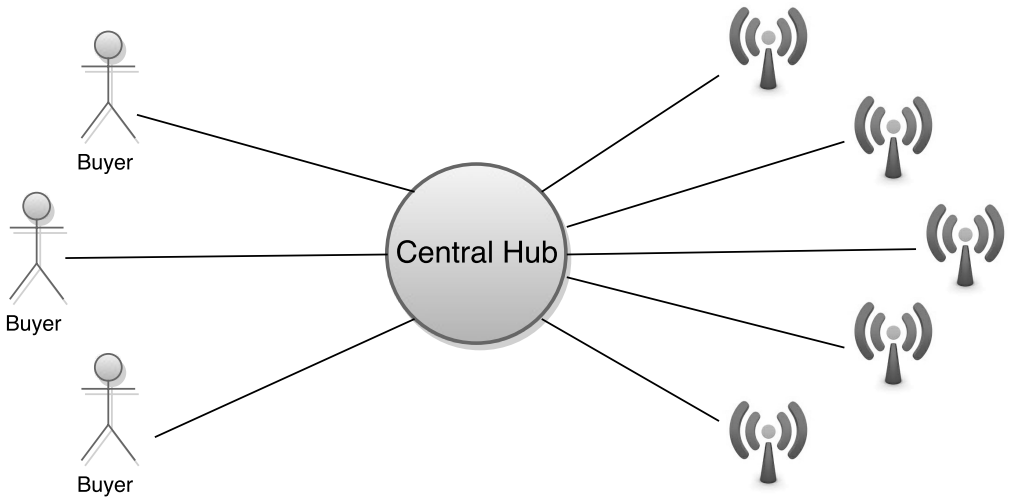


Figure 2.1: System overview: buyers, central hub, and sensor nodes.

2.3 System components

2.3.1 Sensor nodes

The sensor nodes are the data generating elements in the system. Their role is to obtain various parameters about the surrounding environment and provide this data to interested entities for a fixed cost.

In order to participate in the system, the sensor nodes need Internet connectivity to establish TCP connections to the central hub. The node then initiates the establishment of a long-term micropayment channel. Once the setup is complete, the availability of the sensors is signaled to the hub, which registers them accordingly for future buyer queries.

2.3.2 Buyers

This component consists of a simple interface connecting to the hub through a TCP connection. First, a micropayment channel is established with the central hub. After this initial step, the buyer can run queries such as:

- Node statistics: offer information on the number of connected sensor nodes.
- Sensor statistics: offer information on the types of sensors that are available for purchasing.
- Select queries: allow buyers to query for a certain type of sensor data and inform them on nodes this data is available on, together with pricing information.

- Buy queries: allow buyers to purchase a set of data for a certain sensor type from a selected node. This query will return the asked data set.

2.3.3 Central Hub

At the heart of the system lies the central hub. According to the requirements, the hub acts as a coordinator, payment forwarding medium and meeting point between the buyers and the data providers. It establishes micropayment channels with all sensor nodes sharing their data and all buyers. On the other hand, each of the other two components only need to keep one micropayment channel open, with the central hub.

When a new data provider connects to the central entity, it establishes a TCP connection and sets up a new micropayment channel between the two. That is, the hub has to lock in a certain amount for a longer time interval (*30* days, for example). Assuming numerous data providers, the hub is required to have a high amount of bitcoins to pre-allocate in each established channel to run the entire system.

To support the two types of connection, the hub is running two servers, providing specialized handlers that communicate with each sensor node and buyer connecting to it. A driver component coordinates the two subcomponents, bookkeeping connected nodes and buyers, and forwards messages between the buyers and nodes.

2.4 Achieving the requirements

2.4.1 Achieving Scalability and Efficiency

In order to achieve scalability and efficiency, there are two important factors that need to be taken into account: the inherent scalability issue of Bitcoin and the number of peer pairs that need to interact to successfully get payments from buyers to the sensors.

Creating new transactions for each small-value payment would have a very high overhead in terms of fees, which reduces the incentive for both the buyers and the sensor nodes to participate in the protocol. A high volume of transactions would also delay the transactions being confirmed on the Bitcoin network. To address these issues, the proposed system will rely on the previously introduced micropayment channels. This mechanism will allow, after a preliminary setup between the parts that wish to transact, to instantly send small amounts of bitcoins off the block chain, bundling and publishing them to the network only at the end of a specified time window. Thus, both scalability of Bitcoin and efficiency of payments are achieved.

One issue that comes with using micropayment channels in the proposed context is that its set-up phase takes a considerable time, approximately 10 minutes, until the bond transaction is confirmed by the network. Thus, setting up a channel between all buyers and nodes becomes very inefficient time-wise and causes a lot of overhead on the network since all connections need to be maintained for the lifetime of the channel. On the other hand, if the buyers and the sensors have previously established a channel with a central entity as in the hub-and-spoke model [17], each will have a single set-up to wait for and a single channel connection to maintain.

Last, adding a single central entity may affect the scalability of the system when the number of data providers and buyers increases, becoming a performance bottleneck. However, the concept of a central hub should be rather treated as a central cloud, which can contain several machines maintaining the connections, channels, available sensor data, and load balancers that distributed the work across the available machines.

2.4.2 Achieving Security

To achieve efficient end-to-end security on the path between the buyers and the data providers, pre-established micropayment channels are combined with HTLCs. This ensures that all nodes on the payment path receive their funds and no coins can be gained by defecting. The method is first presented for neighboring nodes, then extended for full paths.

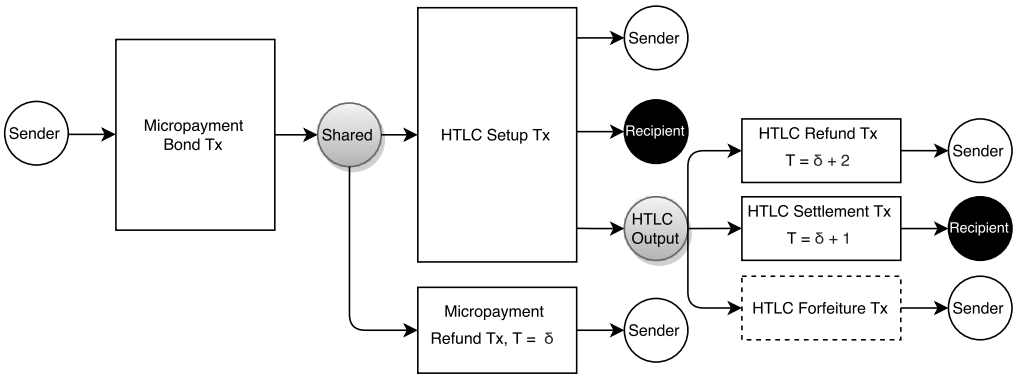


Figure 2.2: Micropayment channels with HTLCs for end-to-end security on payment paths

First, the two neighboring nodes have to set up a shared account (multi-signature output) between the two of them. Once the bond transaction with the multi-signature output is broadcast to the network, the funds are locked in and the micropayment channel is opened. Initially, an HTLC setup transaction spending the shared output is created, allocating the full channel amount to

the sender, and 0 to the recipient. When a new payment is created, instead of directly increasing the amount which the recipient, an HTLC output with the payment amount is added to the HTLC setup transaction, which requires a secret R, known only by the recipient, to be revealed in order to claim the payment. As previously explained in 1.1.3, the HTLC output is spent by three transactions: the HTLC refund, settlement and forfeiture transactions. These transactions are only to be used for conflict resolution.

In a successful collaboration between the two nodes, the recipient will release the secret to the sender. At this stage, an updated HTLC setup transaction is created, with the HTLC output's value added to the receiver's output. Should the recipient not be able to reveal R, it can also choose to void the HTLC output, cancelling the transfer and returning the HTLC output's value to the sender's output of the HTLC setup transaction. The recipient also hands over a forfeiture transaction that guarantees it will not use an older HTLC setup transaction version, should it receive the secret at a later time. Multiple HTLC outputs can be active at the same time and new payments can be made without any transaction being broadcast to the network.

If the sender does not collaborate to update the HTLC setup transaction, the receiving party has to broadcast the most recent HTLC setup transaction before the channel refund transaction becomes valid at time $T = \delta$. If the sender still tries to broadcast the channel refund transaction, it would be rejected since it double-spends the multi-signature output of the bond transaction. Once the HTLC setup transaction is confirmed by the network, the remaining HTLC outputs become available for spending. If the recipient has the secret, it has time until $T = \delta + 1$ to commit the settlement transaction. On the other hand, if the sender cannot produce R by time $T = \delta + 2$, the sender will get its funds back using the HTLC refund transaction. Should the receiver broadcast an older HTLC setup transaction with HTLC outputs that it agreed to void, the sender can broadcast the previously received forfeiture transactions to immediately claim those outputs. Last, if the receiver party vanishes at any point before broadcasting the HTLC setup transaction, the sender can simply use the channel refund transaction to get a full refund of the money it locked into the channel.

To achieve end-to-end security on the full path between the buyers, central hub, and sensor nodes, the protocol above is extended. First, the buyer and the sensor node have to establish micropayment channels with the central hub. In order to make a payment, the buyer has to first make a request to the final recipient, the sensor node, through the hub, which generates a secret R, hashes it to obtain H, and sends H back to the buyer. Using the received hash, the buyer opens a new HTLC flow with the central hub, by adding an HTLC output to the setup transaction spending from the shared account between the two. Once the process is complete, the hub then uses H to open a corresponding HTLC flow with the sensor node. After this step, the sensor can release the secret to

the hub to claim its money. The hub can then use it to claim the funds from the buyer, closing the two HTLCs. This process provides end-to-end security on the payment path.

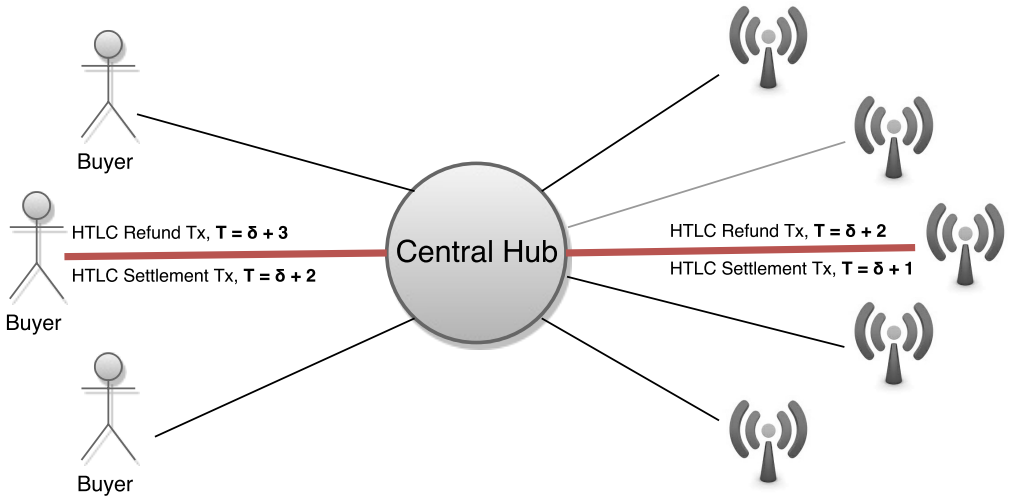


Figure 2.3: Securing payment paths between buyer, hub, and sensor nodes with micropayment channels and HTLCs

In order for the protocol to be fully secure in this two-hop payment scenario, the time locks of the HTLC refund and settlement transactions from the recipient up to the buyer need to be higher than the ones from the previous hop (see 2.3), such that no participant can reveal the secret so late the upstream is not given the opportunity to claim its coins in time.

2.4.3 Achieving Anonymity

By design, the role of the central hub is to connect data buyers and sellers, and given that the lifetime of the established micropayment channels are rather long, the hub can easily analyze the data a buyer is interested in, build a profile and ultimately, identifying it. To address this privacy issue, buyers could make use of an anonymization network, such as Tor, open several, shorter-lived sessions, over several vantage points, and spread payments and data exchanges on these session when buying different sets of data. An additional precaution is to never reuse addresses across sessions. This way, profiling becomes more difficult, and the hub cannot link anymore buyers and data they have purchased through the system.

Chapter 3

Implementation

The focus of this work is to propose a centralized, low-trust, and secure scheme that enables IoT sensor nodes to share their data in exchange for bitcoins. Following the introduced design, the Java implementation of a proof-of-concept based on the Android built-in sensor is presented in this chapter.

3.1 API design

The implementation of the system is based on *bitcoinj*, a Java library that allows users to work with the Bitcoin protocol. Its features include maintaining a Bitcoin wallet, creating, sending and receiving transactions, but also more advance constructs such as contracts and micropayment channels.

From a high-level point of view, the system is designed according to the client-server architecture, in which each component is either a client or a server. The buyer component and the sensor subcomponent of the hub that are spending funds in exchange for a service are considered clients. On the other hand, the buyer subcomponent of the hub and the sensor node component providing data in exchange for bitcoins are considered servers.

From a logical point of view, each application relies on a layered design, which improves maintainability (easy to modify, test, debug, reuse layers), and performance. Layers exchange information through strict interaction. This approach offers a rigorous separation of concerns and ensures that each layer only interacts with the one directly below. Each application is layered according to Figure 3.1:

- **Driver application layer.** This is the topmost layer representing an interface the users can directly make use of to access the functionality provided by the lower layers. Following the design patterns from the *bitcoinj* library, the ListenableFuture class from the Google Guava library is used to register callbacks in the Driver layer, so the application gets notified once a computation is complete.

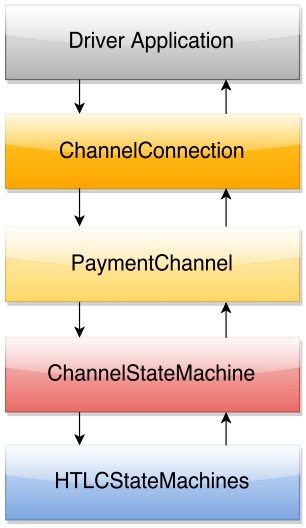


Figure 3.1: Five-layered application architecture

- **Channel Connection layer.** This layer is responsible for reading and writing objects to the network. The functionality is provided by the *niowrapper* package of the *bitcoinj* library, which relies on Java NIO: non-blocking, socket I/O. It also provides multiplexing, a technique used to monitor multiple I/O operations executing concurrently from a single thread, obtaining high performance and low latency.
- **PaymentChannel layer.** The middle layer is responsible for processing the messages that are received through the wire from the layer above, dispatching tasks to lower layer accordingly, receiving results from them, and finally constructing message replies and sending them to the connection layer. Simple micropayment channels are fully supported by the *bitcoinj* library; however, for this project, the package was rewritten to support HTLCs and simplified by removing the layer responsible for resuming channels in case of connection failure.
- **ChannelStateMachine layer.** This layer provides a state machine independent of any network protocol. It is the Bitcoin-aware layer: it constructs transactions, signs them, schedules refunds for future broadcasts to initiate, use, and safely close a micropayment channel.
- **HTLCStateMachines layer.** This layer provides one or more state machines that are responsible for keeping track of the HTLC flow. By allowing several instances to run at the same time, concurrent payments are possible.

Both clients and servers adhere to the architecture above, with differences in functionality. In the next subsection, the two component types are introduced

in more detail. Their interaction to run the proposed payment protocol is then described.

3.1.1 Client

The client component represents the payment sender party in the protocol. The responsibilities of each layer in this component are described in a top-down manner.

At the driver application layer, the client initiates and maintains a connection to the Bitcoin network, synchronizing and keeping the local block chain up to date. It also loads the wallet of the client, and establishes a TCP connection to the server it will send payments to. Once the connection with the server is set up, the micropayment channel establishment protocol is run.

The second layer handles the TCP connectivity. It also acts as a binder between the upper and lower layers and passes messages received from the server to the PaymentChannel layer. The PaymentChannel layer analyzes all incoming messages and triggers specific processes and transitions in lower layers, ultimately constructing appropriate replies that are sent back to the server.

The client uses the ChannelStateMachine layer to store the progress of establishing a micropayment channel so it is always in a secure state. Once the channel is successfully opened, the client can start making HTLC payments. The states of the HTLC payments are stored at the lowest level, in the HTLCStateMachines layer. A full client class diagram can be found in the Appendix, Figure A.1.

3.1.2 Server

The server component represents the payment receiver party in the protocol. The responsibilities of each layer in this component are described in a top-down manner.

At the driver application layer, the server first connects to the Bitcoin network and synchronizes its local block chain. After syncing the block chain, it starts listening for incoming connections from clients in the second layer.

When a new client initiates a TCP connection, the server dispatches a handler that runs the micropayment channel establishment protocol with the client. All messages are processed in the PaymentChannel layer and the state of the payment channel is stored in the ChannelStateMachine layer. Once the channel between the client and the server was established, the server can receive payments from the connected client. The states of the HTLC payments are stored in the HTLCStateMachines layer. A full server class diagram can be found in the Appendix, Figure A.2.

3.1.3 Cross-component communication

System components communicate through TCP connections, transmitting Protocol Buffers (protobuf), an extensible, efficient, automated, serialization method introduced by Google in 2008. After defining the structure of the data that needs to be serialized through an interface description language, a compiler generates source code that allows to write and read the data to in a language independent manner.

To support the proposed payment protocol, the needed data structures were defined using protocol buffer message types in *.proto* files. Each such message contains a series of name-value pairs, associating scalar data types with field names. Additionally, each field has an integer field that uniquely identifies it. The description language also allows specifying field rules:

- *optional*: a message containing a field with this rule can contain this field zero zero or one time.
- *required*: a message containing a field with this rule must contain this field exactly one time.
- *repeated*: a message containing a field with this rule can contain the field multiple times (including zero).

Following the pattern endorsed in the *bitcoinj* library, messages are extended to support the communication in the proposed protocol. A full definition of the used protobuf messages can be found in Appendix A.1. The next subsections will detail out the use of each message in the micropayment channel setup and HTLC payments.

3.1.4 Establishing a micropayment channel

Once the TCP connection between the client and the server components is established, the client will initiate the process to establish a micropayment channel between the two. The exchange of messages in this sub-protocol is asynchronous to increase throughput and response time.

Figure 3.2 shows the steps and exchange of messages between the client and the server to successfully set up a payment channel. The logical states of the channel establishment are displayed in Figure 3.3. The transitions correspond to receiving specific messages from the other protocol participant. First, the client sends over a ClientVersion message, containing the version of the *bitcoinj* library it is using, and the time window in seconds the channel should be opened for. The server verifies if the client version matches its own version, and replies with a ServerVersion message. At this time, the server also sends the client an Initiate

Figure 3.2: Micropayment channel sequence diagram.

message containing its public key, the expire time of the channel, the minimum value that needs to be locked into a channel, and the amount of money that the server needs in the initial payment for the channel to be opened. The last entry ensures that the client does not open a channel that cannot be settled because the server received an amount under the dust is an amount that is below the limits for a legitimate transaction.

When the client receives the Initiate message, it checks that its wallet contains enough funds to open the channel under the requirements imposed by the server, and stores the progress of the channel establishment. The client then creates the bond and the refund transactions. A ProvideRefund message containing the refund transaction and the public key of the client is sent to the server. At this point, the server also stores the progress the channel setup, signs the received refund transaction and returns it to the client in ReturnRefund.

Once it has received the signed refund transaction, the client checks the signature of the server. After validating it, the refund transaction is stored and scheduled for broadcast at the time the channel expires. The client then creates an initial signed HTLC setup transaction spending from the contract transaction a minimum value required by the server, and sends both signed transactions to the server in a HTLCProvideContract message.

At this point, the server checks the signatures of the client and that the HTLC

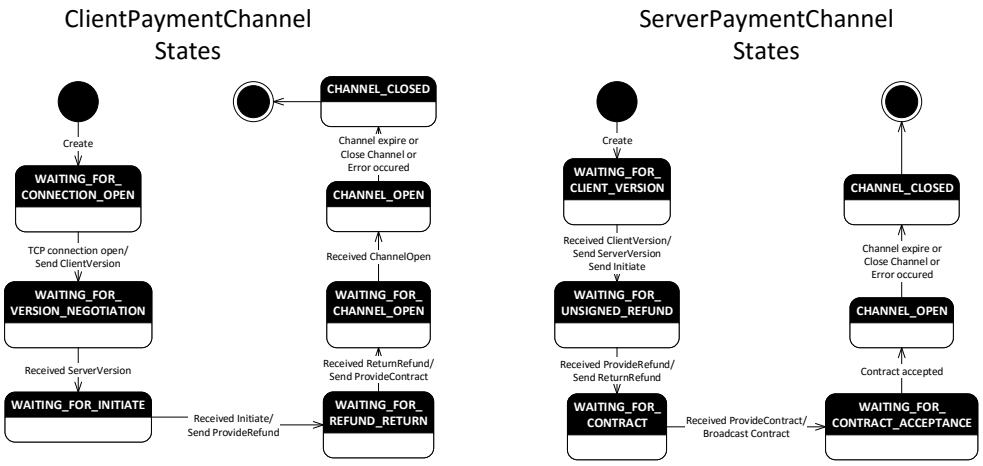


Figure 3.3: Micropayment channel state machine diagram.

setup transaction indeed correctly spends from the contract. After validation, the contract is broadcast to the Bitcoin network. Once the contract is accepted by the network, the funds of the client are locked in, and the server signals the client that the channel is open for payments through a ChannelOpen message.

3.1.5 Making an HTLC payment

After a micropayment channel is successfully set up between the client and the server, the client can begin making payments in exchange for data. When it comes to payments, cross-component communication is no longer done asynchronously, but synchronously, using batched update rounds. This solution has been chosen to support concurrent payments and is needed so that the HTLC setup transaction is always in a secure and consistent state on both the client and server side. Otherwise, updates from both sides could be crossing each other and invalidate the signatures of the HTLC refund, settlement, and forfeiture transactions attached to the HTLC outputs of the setup transaction.

Batching and round negotiation

Figure 3.4 shows how update rounds are negotiated between the client and server. On the client side, updates are always new payments coming in. On the server side, updates can be of two types: the server is either claiming an HTLC output by revealing the corresponding secret, or wishes to void an HTLC output. Both protocol participants buffer update requests in a blocking queue while an update round is active.

When one of the components has a new update to push to its peer, it first

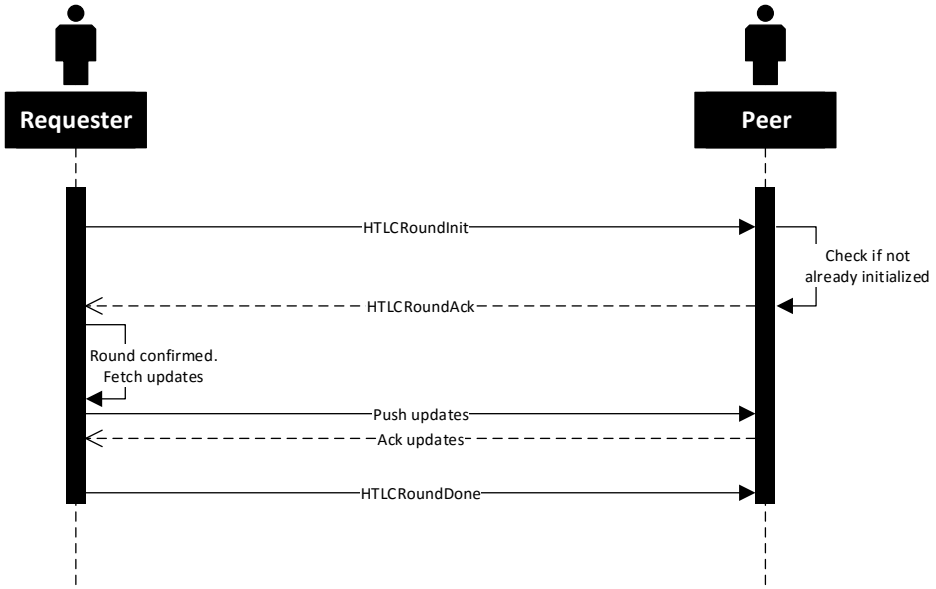


Figure 3.4: HTLC update round negotiation

sends an HTLCRoundInit message to the other participant. Then, the peer checks if no other update round has been already initialized. If this is the case, an HTLCRoundAck is sent to the initiator, which confirms the updates can be pushed. In case both participants are waiting for an HTLCRoundAck message, priority is given to the server. Once the initiator finished pushing all updates to the other participant, it marks the end of the round with an HTLCRoundDone message. If the other participant has buffered updates at this point, it can now safely initiate its own update round.

Initiating a payment

After the client has secured an update round through the negotiation mechanism presented above, it can send new payments to the server. The full process is shown in Figure 3.5. The logical states of the payment setup and clearing are displayed in Figure 3.6. The transitions correspond to receiving specific messages from the other protocol participant.

The client first sends an HTLCInit message containing a list of payments that were buffered during a previous active round. Each payment message contains a client request id, the value of the payment and information about the type of data that is requested in exchange. The server creates new HTLCServerStateMachine

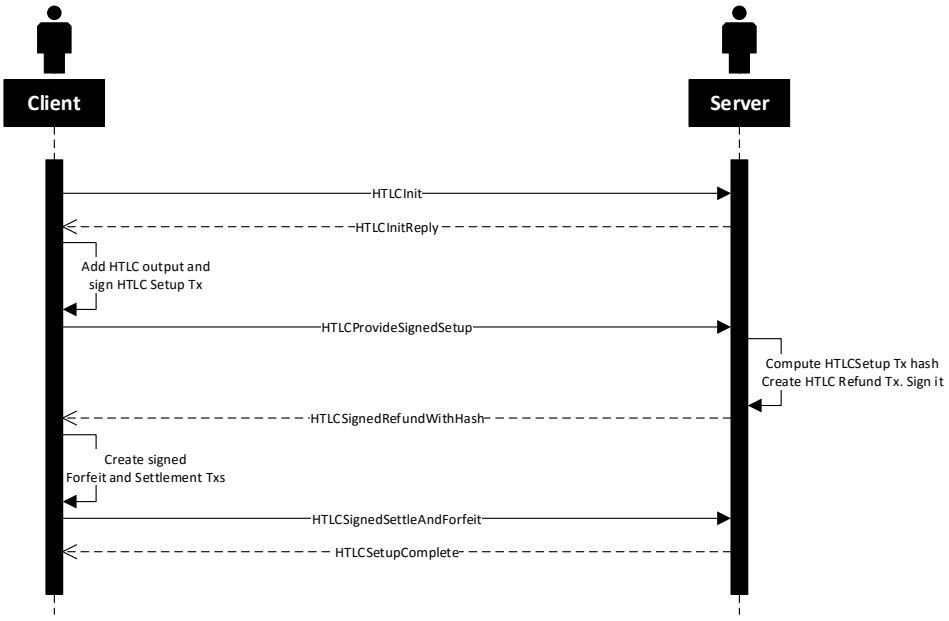


Figure 3.5: HTLC Setup sequence diagram

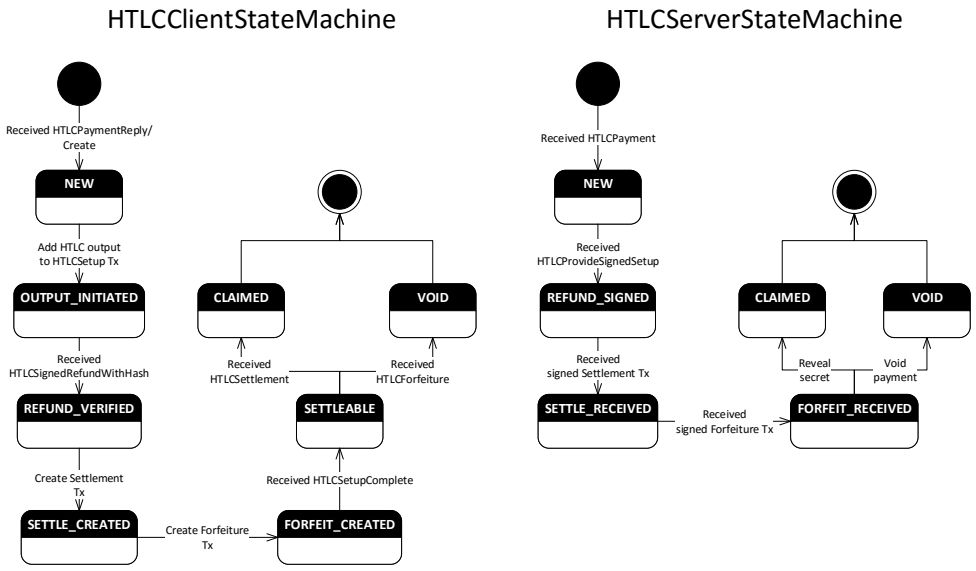


Figure 3.6: HTLC State Machines for both client and server side

instances for each payment. Each HTLC is given an id, which is basically the hash of the secret. An HTLCInitReply message is then replied, mirroring back

the client request id, and sending the fresh HTLC id. From this stage on, each HTLC payment will be referenced by the server-generated id.

At this step, the client can also create HTLCClientStateMachine instances and update the HTLC setup transaction, adding outputs for each initiated payment. The transaction is then signed and sent over to the server, together with the payment ids and indexes of each payment output.

Because of the Bitcoin malleability issue, this stage in the protocol has a major caveat: it sends over the signed HTLC setup transaction before having any signed HTLC refund transaction that protects the client in case the server decides to lock away the funds in the HTLC outputs indefinitely. It cannot be avoided without the malleability fix, because the client cannot create by itself an HTLC refund transaction that spends from the HTLC output: the HTLC setup transaction spends from the multisig output between the two parties, and it needs to be fully signed for the refund transaction to correctly reference it by its hash. Thus, in the current implementation, a minimal amount of trust is needed in the server component, that it will indeed sign, compute the hash of the HTLC setup transaction, and return signed HTLC refund transactions for all active HTLC payments (HTLCSignedRefundWithHash). This step is necessary because removing outputs from the HTLC setup transaction invalidates the signatures of previously created HTLC refund, settlement and forfeiture transactions. The amount of trust can be adjusted by limiting the number of active HTLC payments at a time.

Once the client received the HTLC setup transaction hash and HTLC refund transactions for each payment, the protocol is back to a secure state. The client stores the HTLC refund transactions and broadcasts them when their time lock expires if the HTLC setup transaction makes it to the block chain and the corresponding HTLC outputs are not claimed by that time. The client can now recreate the HTLC forfeiture and settlement transactions for all active HTLC payments, sign and hand them over to the server. After the server checks the signatures and stores the received transactions, it signals the client that the HTLC setup is complete.

Claiming a payment

After a successful setup, the server can initiate an update round to claim the payments from the HTLC outputs. To do so, it first sends an HTLCServerUpdate message to the client. It contains a list of HTLCSettlement and a list of HTLCForfeiture messages. Each reveal message indicates the id of the HTLC payment the server is claiming, together with the secret that hashes to its id. HTLCForfeiture messages contain the the ids of the HTLC payments that are voided and the fully signed HTLC forfeiture transactions, which guarantee the client that an older HTLC setup transaction will not be broadcast, should the

secret be received at a later time. They are sent by the server a fixed time before the channel expiration time and before the HTLC setup transaction is broadcast, such that the number of HTLC refund transactions that are later broadcast by the client is minimized. Thus, the HTLC refund transactions are kept off the block chain, and the server voluntarily opts to void the HTLC payments because it has not received the secret in time.

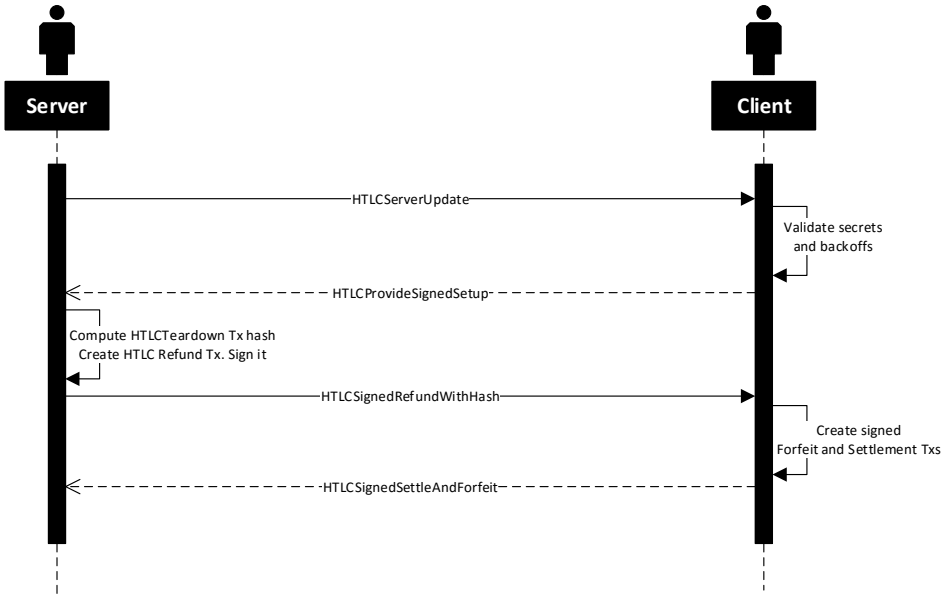


Figure 3.7: HTLC Settlement and Forfeiture sequence diagram

The client validates the received secrets and forfeiture transactions, and removes the HTLC outputs from the HTLC setup transaction accordingly. Forfeiture transactions are stored and hashes of older HTLC setup transactions are followed on the block chain. Should any of them make it to the block chain, the client will automatically broadcast the corresponding forfeiture transactions to get the funds back. The updated HTLC setup transaction is then signed sent to the server using the HTLCProvideSignedSetup message. From here on, the process follows the same steps as in 3.1.5. The message succession is shown in Figure 3.7.

Closing the channel

If the client wishes to close the payment channel before its expiration time, it can send a close message to the server. Then the server can broadcast the most recent HTLC setup transaction to the network, spending the bond trans-

action's output. If the server wishes to close the payment channel early, it simply broadcasts the most recent HTLC setup transaction and signals the client that the channel was closed using the same close message. Should the HTLC setup transaction still contain HTLC outputs, the server can claim them if it has the corresponding secrets using the HTLC settlement transactions, or, if not, the client will get those funds back using the HTLC refund transactions.

Data transmission

In order to securely transmit the data that the client is issuing payments for, the client and the server need a private, authentic out-of-band channel. In the HTLC payment setup phase, the data, encrypted with the HTLC secret, would be transmitted to the client over the hub. Then, once the server has cleared the payment by revealing the secret, the client can decrypt the previously received data. This ensures an atomic exchange of data for bitcoins.

3.2 Application implementation

Following the design introduced in the previous chapter and the client-server architecture described in the previous section, three different applications were developed, corresponding to each of the three components: buyer, central hub, and sensor node represented by an Android application on a device with a wide range of built-in sensors. In the next subsections, each application is presented in detail.

3.2.1 Buyer application

The buyer component was developed as a Java console application. From an architectural point of view, it plays the role of a client in the payment protocol by connecting, initiating a micropayment channel with the hub and making payments to the hub in exchange for data.

Running queries

From the buyer driver console application, the following queries are available:

- Node statistics. *STATS NODES* will display the number of connected sensor nodes with list of sensor node ids.
- Sensor statistics. *STATS SENSORS* will display the types of sensor data that are available for purchasing.

- Select queries. *SELECT SENSOR=<sensor_type>* will display all available nodes with the selected sensor type, together with id of the node it is available on, and pricing information,
- Buy queries: *BUY <sensor_type>FROM node_id price* will purchase the set of sensor data of the specified type from the node selected by id. This query will return the asked data set.

3.2.2 Hub application

The hub application was developed as a Java application. It represents the discovery, coordination and payment forwarding point of the developed system, with both a receiving (server) component from the perspective of the buyers, and a sending (client) component from the perspective of the sensor nodes.

To accommodate this design, the driver application, ChannelConnection, and PaymentChannel layers were modified. A class diagram showing the modified layers can be found in the Appendix, Figure A.3. The ChannelStateMachine and HTLCStateMachines layers were integrated as originally designed.

At the application level, the hub establishes the connection to the Bitcoin network and synchronizes its local block chain. In the ChannelConnection layer, two servers are run on two different ports for: one for connecting buyers, and another one for sensor nodes. The layer also stores data such as mappings of sensor types to the devices that they are installed on, and pricing information, such that it can support node, sensor, and select queries. When a new buyer connects to the hub, a buyer-specific connection handler is used to establish a micropayment channel and receive payments from the buyer. On the other hand, the sensor-specific handlers were adapted to act as servers connection-wise, but as clients in the payment protocol: they initiate micropayment channel establishment with the sensors and send payments to them.

A routing component keeps track of all payment ids and of the buyers and sensor nodes these payments are initiated between. This way, the destination of messages that need to be forwarded between the buyer and the Android components can be easily identified. If a message contains sub-messages for multiple destinations, the router does deep-packet inspection, aggregates the sub-messages by destination, and finally forwards them.

3.2.3 Android application

The sensor node component was developed as an Android application, allowing the user of the app to share sensor data in exchange for bitcoins. Since it represents the receiving party in the protocol, it follows the server component design.

Figure 3.8: Android application user interface.

The user is given full choice regarding which data is shared to the interested buyers. The available device sensors are detected at application start-up. The interface (Figure 3.8) consists of a main view that allows the user to toggle between sharing or not sharing the data of each sensor.

The Bitcoin wallet is created on the first start and locally stored on the device. At this time, the user can also set up prices for sensor sharing. Later on, the user can always go back and set new prices using the settings option. At the top of the interface, the application displays the amount of bitcoins earned so far.

The permissions needed by the developed Android application in order for the device to successfully participate in the built system are specified in the Appendix A.2.1.

ChannelConnection layer adaptation

In the original design, the ChannelConnection layer of the server is the one listening for new connections from connecting clients. However, in this case, the Android application should be the one initiating the connection and registering to the central hub. In this regard, the ChannelConnection layer was adapted such that app is the one initializing the connection (connection client), but the payment protocol message exchange stays the same: the Android component is the receiving party, thus a payment server component.

Connectivity

To establish and maintain the TCP connection to the Hub, the code responsible for the Bitcoin operations was integrated into an Android remote service that starts at device boot-up. This ensures that the long-running operations do not block the application user interface and that the micropayment channel is kept alive even if the user closes the application.

When the service successfully establishes the TCP connection and sets up a micropayment channel with the hub, the application is signaled. Then, the application registers the shared sensors to the hub. This is repeated every time the user changes its preferences in terms of the sensors data that is shared.

When the buyer requests some data from a specific sensor through the hub, the remote service can retrieve the data from the sensor listeners running in the application. Sensor listeners cache the data to allow instant data retrieval. A data container customizes the number of data entries that are cached. The current implementation uses a EvictingQueue of a fixed size K, automatically maintaining the K most recent entries.

3.2.4 Running the full-system protocol

In the full, three-component system, micropayment channels between the participants are established independently. Once at least one buyer and one Android device are connected and channels are in place, the buyer can initiate payments and receive the requested data in return. All requests are made during batched update rounds.

Figure 3.9 shows the message succession needed for the HTLC payment setup. The hub acts as a transparent forwarding medium, passing through the messages between the buyer and the Android device and adjust its state accordingly. First, the buyer sends an HTLCInit message. Upon receiving it, the hub forwards the HTLCInit to the Android device, which replies with an HTLCInitReply message. At this stage, the hub does not continue running the setup protocol with the Android device, but forwards back the reply, finally making it to the

buyer. Now the buyer and the hub can run the HTLC payment setup protocol presented in detail in 3.1.5.

After a successful setup, the hub can resume the paused setup protocol with the Android device. It is important that the HTLC payment is first set up between the buyer and the hub to ensure the security of the protocol: should the hub and the Android device first set up such a payment, nothing guarantees the hub that it can claim the money from the buyer after it has paid the Android device.

When the Android device claims or voids some HTLC payments, it follows the exact pattern introduced in 3.1.5. Only this time, after the hub receives and validates the HTLCServerUpdate message, it forwards it immediately to the buyer, and then continues with its update round. This way, the hub can run the update protocol simultaneously and the buyer can use the secret to decrypt the received data that was included in the HTLCInitReply message.

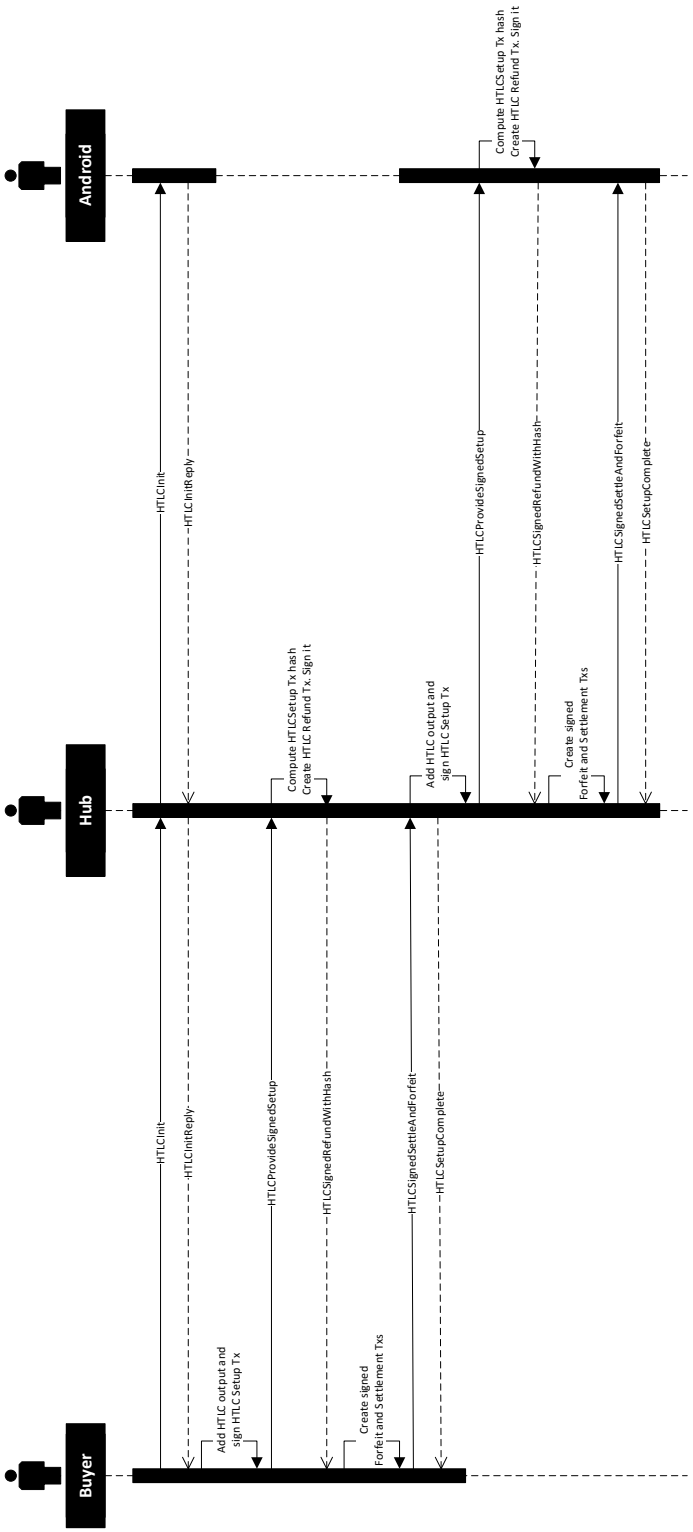


Figure 3.9: HTLC Full Setup sequence diagram

Chapter 4

Evaluation

In this chapter, the system is evaluated both in terms of performance and security threats.

4.1 Performance

The performance of the built system was evaluated on the Bitcoin TestNet. The TestNet uses a different block chain with coins that have no value, making it an attractive platform for testing applications and experimenting with no financial loss.

4.1.1 Setup

For the performance evaluation, the following setup was used:

- **Central hub**. The hub was given 1000 bitcoins.
- **Android component (Samsung GT-I9300)**. The component was connected to the central hub and registered with two sensors: *light* and *proximity*. It starts with an empty wallet.
- **Buyers**. 100 buyers were connected to the hub. Each was given 1 bitcoin.

The central hub and the buyer application were run on the same local machine. All three components were connected to the same local Internet network.

Micropayment channels

The time needed to set up the micropayment channels between the buyers and the hub was measured. In the *bitcoinj* library, the server signals the client that the micropayment channel has been opened immediately after the multi-signature contract propagated to the network. However, in order to be truly

secure, the channel should only be considered open after the transaction was included in a block in the block chain. Measurements were made according to this observation.

Making queries

Each buyer was set up such that it chooses one of the four available queries with equal probability every 5 seconds, putting a constant load on the system:

- *STATS NODES*.
- *STATS SENSORS*.
- *SELECT SENSOR=<sensor_type>*. Where <sensor_type>is randomly chosen between the two available sensors.
- *BUY <sensor_type> FROM node_id price*. The price was fixed to *1* micro-bitcoin.

The following data was collected during the experiment:

- *Response times* for each query type.
- *HTLC setup times*. The time needed to set up the HTLCs was measured both between each pair of components, and on the complete path.
- *HTLC resolution times*. The time needed to tear down the HTLCs and restore the HTLC setup transaction to a secure state was measured.

4.1.2 Results

Micropayment channels

As it can be seen in Table 4.1, the time spent by buyers to establish a micropayment channel with the central hub varies from a couple of seconds to just under a minute. This variation is given by the probabilistic nature of proof of work performed by the miners to include transactions in new blocks. In addition, the TestNet tends to be rather unstable due to experimental work. In the main Bitcoin network, the expected time to get a transaction included in the block chain is 10 minutes.

	Micropayment channel
Mean	22.114 [s]
St. Dev.	13.588 [s]

Table 4.1: Micropayment channel setup time average and standard deviation on TestNet.

Queries

Query type	Nr. of queries	Mean	St. Dev.
Select	853	41.62 [ms]	2.86 [ms]
Node stats	863	40.91 [ms]	2.68 [ms]
Sensor stats	922	40.75 [ms]	4.25 [ms]
Buy	744	1581.74 [ms]	1497.95 [ms]

Table 4.2: Mean and standard deviation of the response times for select, node and sensor queries.

To get a better understanding of where is most of the response time for buy queries is spent in the system, a breakdown of the measurements is given for each hop on the payment path.

Buy queries breakdown	Mean	St. Dev.
Full setup	1054.56 [ms]	151.40 [ms]
Buyer-Hub setup	479.29 [ms]	78.81 [ms]
Hub-Sensor setup	1015.08 [ms]	151.41 [ms]
Hub-Sensor teardown	199.60 [ms]	68.29 [ms]
Buyer-Hub teardown	146.57 [ms]	10.09 [ms]

Table 4.3: Buy query response time breakdown by payment hop.

In Table 4.3, the HTLC setup time, which includes initiating a payment, updating the HTLC setup transaction and the exchange of HTLC refund, settlement, and forfeiture transactions on the full path, is shown.

Discussion

An important observation regarding the performance results is the high transaction confirmation times on the Bitcoin network. On average, the Bitcoin TestNet took 22.12 [s]. However, on the main Bitcoin network, it takes about 10 minutes for a transaction to be included in the block chain. If each payment in the system was included in an individual transaction, the overall experience of the buyer would be poor, encountering high fees and query times, and possibly

getting transactions rejected by the system because of the anti-flood algorithms running on the network.

On the other hand, micropayment channels only require one transaction to be confirmed, locking in the funds allocated by the buyer. After this setup time, queries and payments can be made securely and very fast, without broadcasting any transaction to the block chain.

By analyzing the obtained performance results, it can be observed that most of the time in a buy query is spent during the HTLC setup phase. In fact, the setup phase takes on average 1054.56 [ms], which represents 67% out of the total buy query time. This is mainly due to the fact that the sensor node has to wait until the HTLC is successfully set up between the buyer and the hub (479.29 [ms] on average), before it can resume the HTLC setup process with the hub. Without this condition, the node cannot be assured that it can claim the created HTLC output from the hub. Thus, the hub-sensor setup phase needs an average of 1015.08 [ms] to complete.

In the HTLC tear down phase, the hub and sensor node took an average of 199.60 [ms], and the buyers and the hub 146.57 [ms] to get back to a secure state in the channel. The performance of the two hops is more similar because this stage is parallelized: after the hub receives the secret from the sensor node, it validates it and immediately forwards it to the buyer instead of waiting to clear the HTLC output from the HTLC setup transaction. When the buyer also received the secret, it can independently update its state with the hub. On average, the tear down stage between the hub and the sensor node takes longer because the latter has a poorer hardware performance.

4.2 Protocol vulnerabilities

In this section, the vulnerabilities of the built system are briefly evaluated and countermeasures for each such vulnerability are introduced.

4.2.1 Flow messages

In the current proof-of-concept, the system transmits all messages between components using the Google Protocol Buffers. An attacker could eavesdrop, modify, or inject messages on the used TCP connections, gathering information and learning about what is happening in the system. Even though the messages are serialized in binary format, the format is publicly-known and can be easily decoded. A countermeasure for this vulnerability is to use socket layer encryption.

4.2.2 Sensor data

Data validity

The introduced protocol guarantees the buyer that it will get the data it paid for. In the payment setup phase, the data encrypted with the secret is transmitted to the buyer. If the data is not received during this stage, the buyer can simply choose not to continue running the rest of the payment protocol and use the channel refund transaction to get its money back.

Assuming that the payment is successful, the buyer will receive the secret to decrypt the data it is in possession of. However, the atomic exchange of data for bitcoins does not guarantee the validity of that piece of data. The data could be corrupt, random bytes containing no real sensor data, or could be simply faked by nodes with no sensors.

In order to mitigate this, it is important to consider a reputation system on the central hub. Since each sensor is registered upon connecting with a unique device id that the buyers can use to get specific sensor data from, the hub could also publish reputation scores for each sensor node on the *select* queries. Then, after consulting the scores, the buyer can make an educated choice regarding the nodes it buys data from.

Data confidentiality

In the built proof-of-concept, the data is transmitted between the sensor nodes and the buyers through the central hub. Since the secret disclosure is done backwards on the full payment path, the hub will also come into possession of the secret and can decrypt the data purchased by the buyer. This violates the confidentiality since the data is disclosed to an unauthorized entity.

A possible countermeasure for this issue is to use onion-encryption. When initializing a new payment, the buyer would transmit its public key to the hub, which then forwards it to the sensor node. At this stage, the node can encrypt the data with the generated secret, and then encrypt it again using the received public key of the buyer. This way, even if the hub attempts to decrypt the upper layer of the data, it cannot fully decrypt it since it does not have the corresponding private key.

Another option is to combine the public key of the buyer with the payment secret. This can be done because the public key cryptography based on elliptic curve is an additive homomorphism and allows adding the secret to the private and public keys separately. Using the combined key, the sensor node can encrypt the data and send it to the buyer through the hub. Since only the buyer is in possession of the corresponding private key, it is the only one able to decrypt the received data.

Chapter 5

Conclusion

Bibliography

[1] Bitcoin discussion forum. https://bitcointalk.org/. Accessed September 2015.

[2] Alex Akselrod. Draft. https://en.bitcoin.it/wiki/User:Aakselrod/Draft, March 2014. Accessed September 2015.

[3] Alex Akselrod. Eschaton. https://gist.github.com/aakselrod/9964667, April 2014. Accessed September 2015.

[4] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A Survey. *Comput. Netw.*, 54(15):2787–2805, oct 2010.

[5] Adam Back. Hashcash - A Denial of Service Counter-Measure. Technical report, 2002.

[6] C.J.Plooy. Combining bitcoin and the ripple to create a fast, scalable, decentralized, anonymous, low-trust payment network. http://www.ultimatestunts.nl/bitcoin/ripple_bitcoin_draft_2.pdf, January 2013. Accessed September 2015.

[7] Christian Decker and Roger Wattenhofer. Bitcoin transaction malleability and mtgox. *CoRR*, abs/1403.6676, 2014.

[8] Mike Hearn and Jeremy Spilman. Bitcoin contracts. https://en.bitcoin.it/wiki/Contracts. Accessed September 2015.

[9] Silvio Micali and Ronald L. Rivest. Micropayments revisited. In *Topics in Cryptology - CT-RSA 2002, The Cryptographer's Track at the RSA Conference, 2002, San Jose, CA, USA, February 18-22, 2002, Proceedings*, pages 149–163, 2002.

[10] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf. Accessed September 2015.

[11] Kay Noyen, Dirk Volland, Dominic Wörner, and Elgar Fleisch. When Money Learns to Fly: Towards Sensing as a Service Applications Using Bitcoin. *CoRR*, abs/1409.5841, 2014.

[12] Rafael Pass and abhi shelat. Micropayments for decentralized currencies.

[13] Magdalena Payeras-Capellà, Josep Lluís Ferrer-Gomila, and Ll. Huguet-Rotger. An Efficient Anonymous Scheme for Secure Micropayments. In *Proceedings of the 2003 International Conference on Web Engineering*, ICWE'03, pages 80–83, Berlin, Heidelberg, 2003. Springer-Verlag.

[14] Joseph Poon and Thaddeus Dryja. The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments. https://lightning.network/lightning-network-paper.pdf, July 2015. Accessed September 2015.

[15] Ronald L. Rivest and Adi Shamir. Payword and micromint: Two simple micropayment schemes. In *Proceedings of the International Workshop on Security Protocols*, pages 69–87, London, UK, UK, 1997. Springer-Verlag.

[16] Xiang Sheng, Jian Tang, Xuejie Xiao, and Guoliang Xue. Sensing as a service: Challenges, solutions and future directions. *Sensors Journal, IEEE*, 13(10):3733–3741, 2013.

[17] Peter Todd. Near-zero fee transactions with hub-and-spoke micropayments. http://sourceforge.net/p/bitcoin/mailman/message/, December 2014. Accessed September 2015.

[18] Daniel Wilusz and Jarogniew Rykowski. Requirements and general architecture of a payment system for the Future Internet. volume 2, pages 91–103, 2012.

[19] Daniel Wilusz and Jarogniew Rykowski. The Architecture of Coupon-Based, Semi-off-Line, Anonymous Micropayment System for Internet of Things. In *Technological Innovation for the Internet of Things*, volume 394 of *IFIP Advances in Information and Communication Technology*, pages 125–132. Springer Berlin Heidelberg, 2013.

[20] Yu Zhang and Jiangtao Wen. An IoT electric business model based on the protocol of bitcoin. In *ICIN*, pages 184–191. IEEE, 2015.

Appendix A

Appendix Chapter

A.1 Protobuf messages

All messages prefixed with *HTLC** are introduced for the scope of this work, while the others already existed in the current version of the library.

```
message TwoWayChannelMessage {
    enum MessageType {
        CLIENT_VERSION = 1;
        SERVER_VERSION = 2;
        INITIATE = 3;
        PROVIDE_REFUND = 4;
        RETURN_REFUND = 5;
        PROVIDE_CONTRACT = 6;
        CHANNEL_OPEN = 7;
        UPDATE_PAYMENT = 8;
        PAYMENT_ACK = 11;

        HTLC_PROVIDE_CONTRACT = 12;
        HTLC_INIT = 13;
        HTLC_INIT_REPLY = 14;
        HTLC_SIGNED_SETUP = 15;
        HTLC_SIGNED_REFUND = 16;
        HTLC_SIGNED_SETTLE_FORFEIT = 17;
        HTLC_SETUP_COMPLETE = 18;
        HTLC_SERVER_UPDATE = 19;
        HTLC_UPDATE_SETUP = 20;
        HTLC_DATA = 21;

        HTLC_ROUND_INIT = 22;
        HTLC_ROUND_ACK = 23;
        HTLC_ROUND_DONE = 24;
        HTLC_FLOW = 25;
```

```
    };

    required MessageType type = 1;

    optional ClientVersion client_version = 2;
    optional ServerVersion server_version = 3;
    optional Initiate initiate = 4;
    optional ProvideRefund provide_refund = 5;
    optional ReturnRefund return_refund = 6;
    optional ProvideContract provide_contract = 7;
    optional UpdatePayment update_payment = 8;
    optional PaymentAck payment_ack = 11;
    optional Settlement settlement = 9;

    optional HTLCProvideContract htlc_provide_contract = 12;
    optional HTLCInit htlc_init = 13;
    optional HTLCInitReply htlc_init_reply = 14;
    optional HTLCProvideSignedSetup htlc_signed_setup = 15;
    optional HTLCSignedRefundWithHash htlc_signed_refund_with_hash = 16;
    optional HTLCSignedSettleAndForfeit
                htlc_signed_settle_and_forfeit = 17;
    optional HTLCSetupComplete htlc_setup_complete = 18;
    optional HTLCServerUpdate htlc_server_update = 19;
    optional HTLCData htlc_data = 21;
    optional HTLCRoundInit htlc_round_init = 22;
    optional HTLCRoundAck htlc_round_ack = 23;
    optional HTLCRoundDone htlc_round_done = 24;
    optional HTLCFlow htlc_flow = 25;
}

message ClientVersion {
    required int32 major = 1;
    optional int32 minor = 2 [default = 0];
    optional bytes previous_channel_contract_hash = 3;
    optional uint64 time_window_secs = 4 [default = 86340];
}

message ServerVersion {
    required int32 major = 1;
    optional int32 minor = 2 [default = 0];
}

message Initiate {
    required bytes multisig_key = 1;
```

```
    required uint64 min_accepted_channel_size = 2;
    required uint64 expire_time_secs = 3;
    required uint64 min_payment = 4;
}

message ProvideRefund {
    required bytes multisig_key = 1;
    required bytes tx = 2;
}

message ReturnRefund {
    required bytes signature = 1;
}

message ProvideContract {
    required bytes tx = 1;
    required UpdatePayment initial_payment = 2;
}

message UpdatePayment {
    required uint64 client_change_value = 1;
    required bytes signature = 2;
    optional bytes info =  3;
}

message PaymentAck {
    optional bytes info = 1;
}

message Settlement {
    required bytes tx = 1;
}

message HTLCProvideContract {
    required bytes tx = 1;
    required HTLCSignedTransaction signed_initial_setup = 2;
}

message HTLCRoundInit {
    optional bytes info = 1;
}

message HTLCRoundAck {
    optional bytes info = 1;
```

```
}

message HTLCRoundDone {
    optional bytes info = 1;
}

message HTLCPayment {
    required string request_id = 1;
    required string device_id = 2;
    required string sensor_type = 3;
    required uint64 value  = 4;
}

message HTLCInit {
    repeated HTLCPayment new_payments = 1;
}

message HTLCPaymentReply {
    required string id = 1;
    required string client_request_id = 2;
}

message HTLCInitReply {
    repeated HTLCPaymentReply new_payments_reply = 1;
}

message HTLCSignedTransaction {
    required bytes tx = 1;
    optional bytes tx_hash = 2;
    required bytes signature = 3;
}

message HTLCProvideSignedSetup {
    repeated string ids = 1;
    repeated int32 idx = 2;
    required HTLCSignedTransaction signed_setup = 3;
}

message HTLCSignedRefundWithHash {
    repeated string ids = 1;
    repeated HTLCSignedTransaction signed_refund = 2;
}

message HTLCSignedSettleAndForfeit {
```

```
    repeated string ids = 1;
    repeated HTLCSignedTransaction signed_forfeit = 2;
    repeated HTLCSignedTransaction signed_settle = 3;
    required bytes client_secondary_key = 4;
}

message HTLCSetupComplete {
    repeated string ids = 1;
}

message HTLCSettlement {
    required string id = 1;
    required string secret = 2;
}

message HTLCForfeiture {
    required string id = 1;
    required HTLCSignedTransaction signed_forfeit = 2;
}

message HTLCServerUpdate {
    repeated HTLCSettlement reveal_secrets = 1;
    repeated HTLCForfeiture back_offs = 2;
}
```

A.2 Android application

A.2.1 Permission requirements

- **android.permission.INTERNET**. The application needs Internet connectivity at all times.
- **android.permission.WRITE_EXTERNAL_STORAGE**. The application needs permission to store and update the wallet and block chain files.
- **android.permission.RECEIVE_BOOT_COMPLETED**. The application needs to receive the BOOT_COMPLETED signal to initialize the micropayment channel to the hub.

A.3 UML diagrams

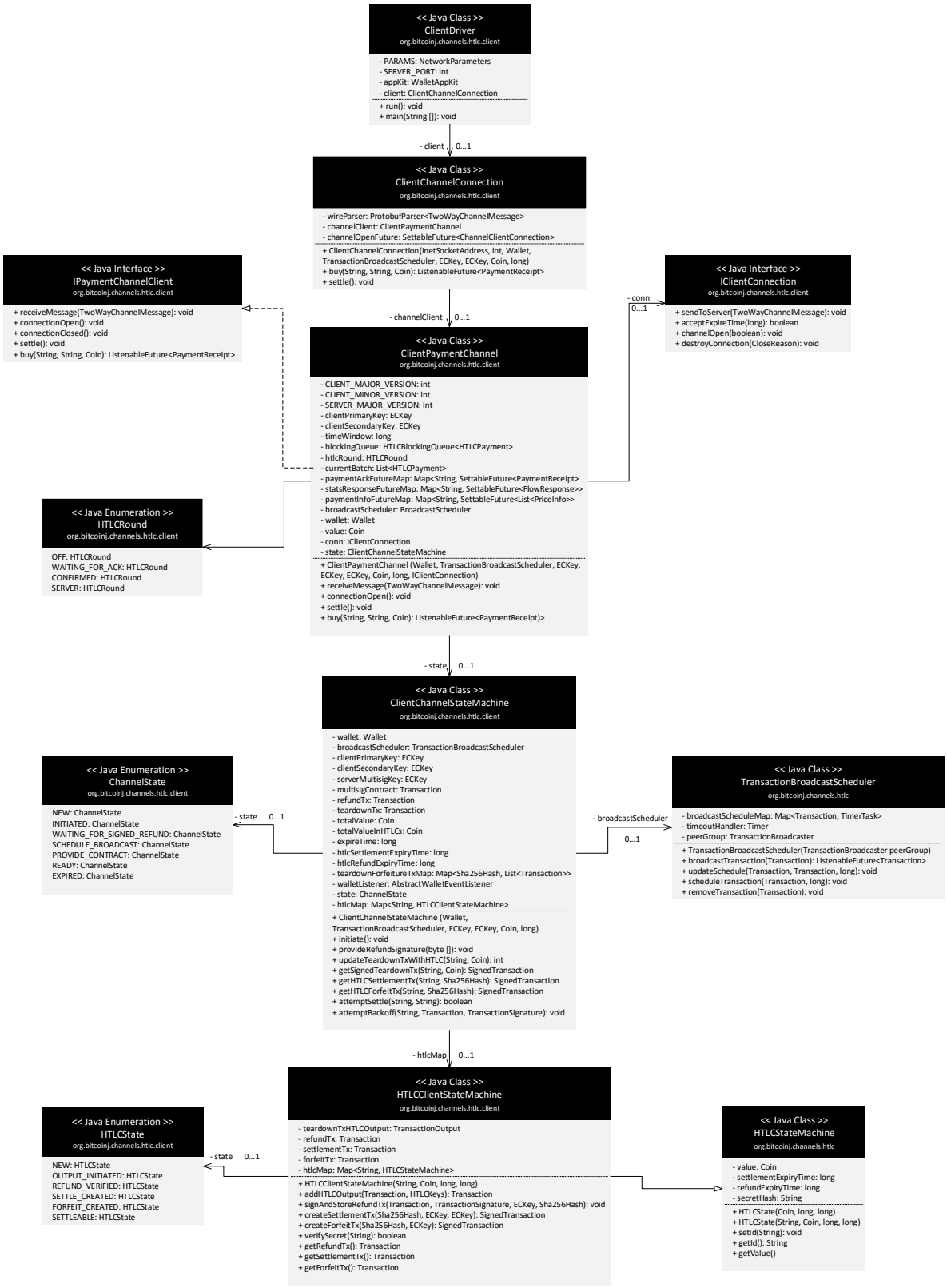


Figure A.1: UML class diagram of the Client component

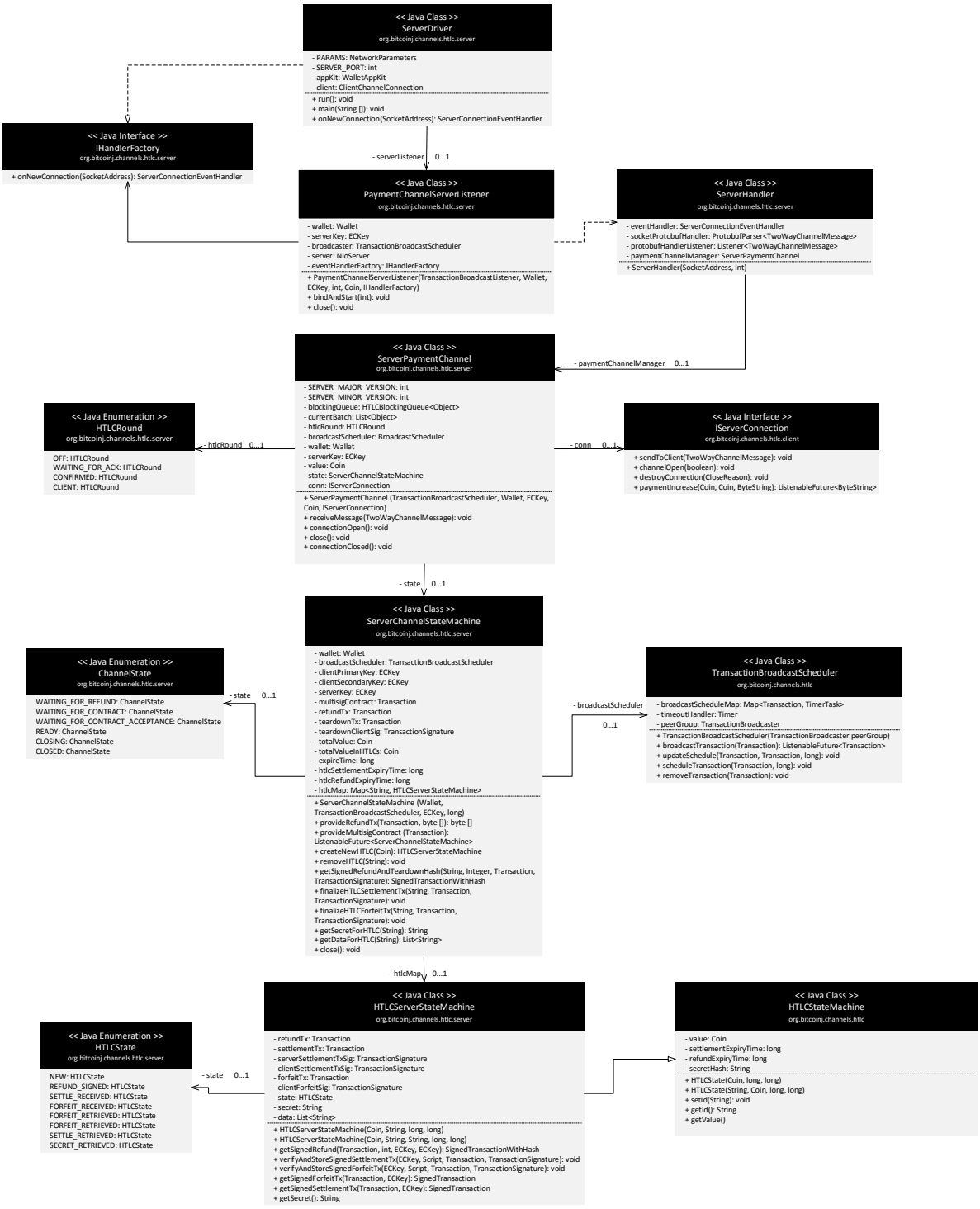


Figure A.2: UML class diagram of the Server component

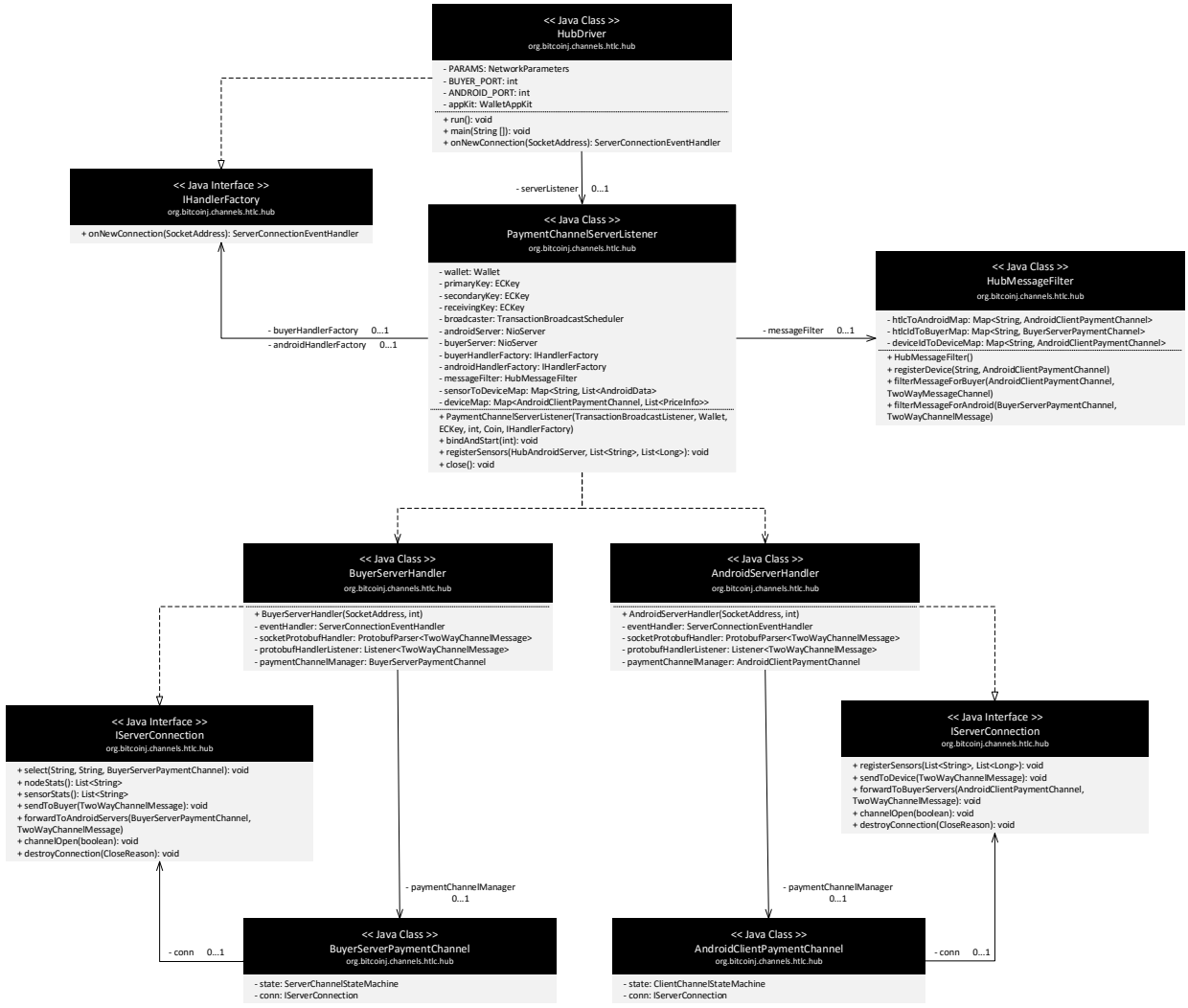


Figure A.3: Hub class diagram of the first three layers.