Design and implementation of NDN-based Ethereum blockchain

Quang Tung Thai[1,*], Namseok Ko[1], Sung Hyuk Byun[1], Sun-Me Kim[1]

Electronics and Telecommunications Research Institute (ETRI), 218 Gajeong-ro, Yuseong-gu, Daejeon, 34129, South Korea

1. Abstract

Blockchain technology allows public parties to agree on a common state without relying on a central authority. Despite it brings many innovative use cases, the technology is still in its early stage that needs improving on many aspects. One of the issues is to deliver blockchain data more efficiently. Named Data Networking (NDN), a new network paradigm, is designed to make content distribution with ease by enabling in-network caching and built-in multicasting, which blockchain technologies can take advantage. Moreover, blockchain may contribute to extending NDN application ecosystems including decentralized applications. Therefore, it is instrumental to have a working blockchain system that runs on NDN platform to supports its research and development.

In this work, we design and implement an NDN-based Ethereum blockchain platform. We propose new protocols for propagating blockchain data making full use of NDN features for the delivery of transactions and blocks. Our experiments show that the distribution of blockchain data in NDN is more efficient than that of IP network. The latency of block delivery is also reduced, which in turn supports tuning blockchain parameter for better security. Our developed blockchain client is going to be freely distributed as an open-source project. We hope that it can provide a platform to foster blockchain research on NDN in the future.

Keywords: *Blockchain, Ethereum, P2P, ICN, NDN.*

2. Introduction

The Internet was originally designed to address the needs of a network for sharing resources among hosts. The basic requirement at that time was to forward data packets among a limited number of nodes. The data exchange is realized by establishing a pairwise communication channel. The simple design based on a host-to-host communication model eases the network expansion,

*Corresponding author

allowing the Internet to grow at a tremendous size. With evolution, an unprecedented number of innovations in term of applications, services, and underlying technologies has emerged. Internet usage has been shifting away from its original design intention for host-to-host communication, leaning toward massive content retrieval and distribution where users request contents from the Internet without caring where they are located. It consequently introduces new requirements for the Internet architecture to fulfill the emerging needs. The mismatching between the host-centric communication model and the content-centric usage makes it difficult to fulfil the requirements and solutions come in patches.

The Information-Centric Networking (ICN)[1], [2] paradigm was born as an attempt to design the future Internet architecture from a clean slate. Viewing the Internet as a content delivery network, the ICN aims to reflect its needs better than the current complex Internet architecture. The basic idea is to view content as basic units of the network instead of identified hosts. In ICN, users only need to express their interests in given contents and the entire network is in charge of forwarding the requests to the best content provider and delivering the contents in the reversed paths. These features are built directly on the network layer by naming contents and implementing all networking activities on their names. Naming content at the network layer allows ICN to support in-network caching and multicast mechanisms natively, thus facilitating an efficient and timely content delivery.

Named Data Networking (NDN) [3] is one of several realizations [4], [5] of the ICN concept. It receives a growing interest from academic and research industry community. There is a large body of research and a growing active code base. Its global testbed network has more than 30 institutions participating.

A blockchain is a distributed database that records the history of digital transactions. What makes the technology interesting is that the database is maintained by public nodes without a central authority. Any interested party can participate in the blockchain system. The nodes run blockchain client applications that communicate to synchronize on a shared ledger. The ledger is structured as a hash chain for data immutability. Cooperation among the nodes is possible thanks to the ingenious design of Proof of Work (PoW) consensus algorithm introduced in Bitcoin network, the blockchain pioneer. From then, Ethereum blockchain was born with a notion of smart contract that turns a blockchain into a world computer which can be run by public. A smart-contract is a Turing complete small program residing in blockchain ledger that can be executed to realize complex logic transactions. Its introduction promises to bring about many innovative use cases.

Blockchain technology may play a vital role in the decentralized Internet. In the beginning, the Internet was designed to operate in a distributed and decentralized manner. Partial removal/addition does not interrupt the system functioning. However, over time the network has evolved to become centralized. At current status, some core Internet applications and infrastructure services are provided by a few big players. This monopoly puts the Internet end-users at risk of losing their privacy. The blockchain can be one important piece of

the puzzle to solve the centralized Internet problems. As ICN is considered the infrastructure for the future Internet, it is foreseeable that blockchain technology is instrumental for supporting decentralized applications.

Unsurprisingly, the applications of blockchain technology in ICN have been growing significantly. The list of topics includes but is not limited to: public key infrastructure [6], data security and access control [7], vehicle network [8], etc. However, all those works are conceptual ideas without practical implementation. Recently there have been some attempts to develop a blockchain system for NDN infrastructure. BlockNDN [9] and BoNDN [10] were developed purely for analysis of block propagation in NDN. Dledger [11] was developed for a specialized application. The lack of a generic practical platform to support blockchain research in ICN community has motivated us to work on the design and implementation of the first NDN-based blockchain system.

Our motivations also come from the interesting question which has been raised in previous works: does blockchain technology fit to ICN? More specifically, we aim to take advantage of the in-network caching and native multicasting features of the ICN to design a better blockchain system where data can be disseminate efficiently. Although there are abundant works on enhancing blockchain technology, the focus has been on the topics of applications, network security, and consensus protocols. There is little research attention on data transport aspect. Much of data in a blockchain system is delivered to every node using a gossip protocol on its peer-to-peer (P2P) overlay which is highly inefficient. This problem of content delivery in public systems is difficult to solve because there is no feasible incentive mechanism for content caching. However, the shifting of underlying network infrastructure from IP to NDN may enable a new design of the data delivery on blockchain networks.

In this work, we aim to develop a blockchain system to foster the blockchain research and deployment of blockchain applications on NDN. Specifically, we design and implement an NDN-based Ethereum client, the single core component of a blockchain network. Instead of starting from scratch, we base our implementation on the existing open-source Ethereum client. The reason for choosing the Ethereum blockchain is due to its well-tested security and popularity in academic research. The Ethereum network has shown its durability and resistance against malicious attacks. Its client implementation was the code base for many other blockchain adaptations. Research on blockchain applications, security, and consensus has been using Ethereum technology and its networks as research materials. For adapting to the NDN environment, its IP-based transport layer has to be replaced with an NDN-based counterpart. For realizing that, we design new protocols for the delivery of blockchain data on NDN and implement them to replace the IP-based implementation.

In contrast to previous works [9],[10], [11] which abandon the P2P overlay for simplistic designs, we argue that even though ICN architecture has built-in features to facilitate efficient content delivery, a practical blockchain system still requires it. Our argument rests on the requirement we set for the target blockchain system. It is a scalable public network: anyone can join or leave the system without notification. It is impractical to address the requirement without

relying on a P2P overlay. We next propose a new approach for data delivery that can take full use of in-network caching and multicasting of NDN without losing the system scalability. The core idea is to utilize the P2P overlay to disseminate announcements of new data items while using the pull mechanism of NDN to fetch them with their unique names. Traffic redundancy caused by gossiping over P2P is lessen due to the small size of the announcements. However, the data items can be cached on the network to enable multicasting; thus, delivering them to all the nodes incurs no traffic redundancy.

In summary, this paper makes following contributions:

- We design a data dissemination protocol for efficient delivery of blocks and transactions of blockchain system in NDN infrastructure.
- We implement the protocol and integrate it into a fully complete first Ethereum client for NDN network.
- We conduct experiments on an emulated network and show that comparing to a similar IP-based system, an NDN-based blockchain network using our implemented client uses much less traffic for ledger synchronization and propagates blocks more quickly.

The remaining of the paper is organized as follows. We explain the basic concepts of NDN architecture and the main blockchain concepts in Section 3. The design and realization of data delivery protocols on NDN is described in Section 4. We next proceeds to the implementation and evaluation of our design in Section 5. We enumerate several related works in Section 6 and conclude our work in Section 7.

3. Background

3.1. Named Data Networking

NDN communication model resembles the request/response scheme in HTTP protocol. In this model, a consumer is requesting a piece of data while a producer is serving it. NDN uses two kinds of packets: Interest and Data. The Interest packet is used for requesting while the Data packet contains the data piece. Both packets must have a name that identifies the data. When a consumer wants to fetch the data, it sends an Interest packet bearing the data name to the network. Network routers use the name to forward the packet to its producer. The producer returns the data in a Data packet of the same name. The packet travels in the reverse path taken by its Interest counterpart.

The format of NDN packets is presented in Figure 1 which is reproduced from Zhang et al. [3]. Of all data fields on an Interest packet, `Name` and `ForwardingHint` are essential for packet forwarding. The `Name` field is to identify requested content. In addition, it may include forwarding information to help network navigate the request to its producer. In the case where the packet is not forwardable based on its name alone, a hint should be provided to the

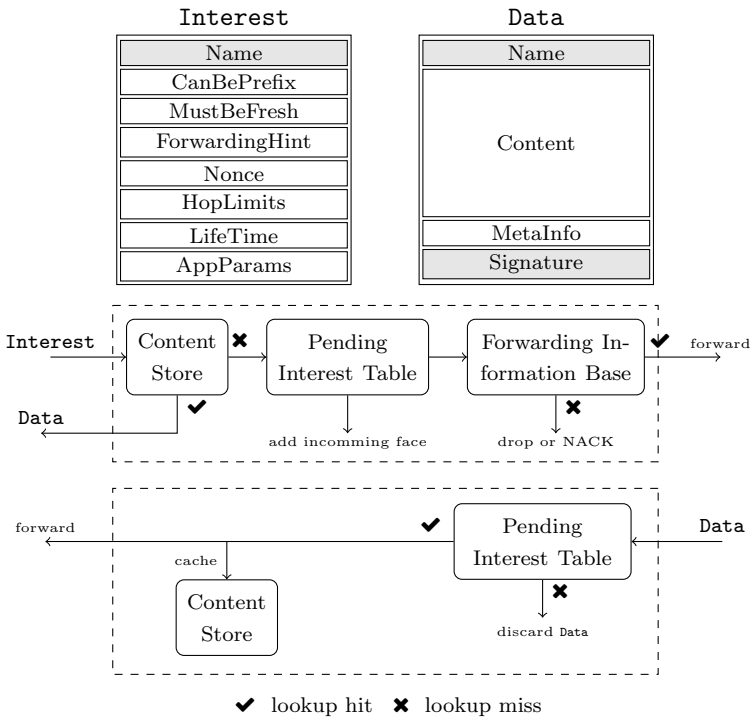


Figure 1: NDN packet format and packet processing at NFD

`ForwardingHint` field for aiding. In typical uses, the `ForwardingHint` is usually the name prefix of a domain where data producer is located. The `Name` value of the packet must be, therefore, routable within the domain.

A Data packet must bear the same name as its Interest counterpart. The `Content` field has the requested data in an application-specific format. As NDN advocates security and privacy protection at the packet level, all data packets should be secured with digital signatures. A producer should verify the integrity and authenticity of the data using the signature (`Signature` element) and its information included in the data packet.

An NDN network consists of routers and hosts. They run an NDN Forwarding Daemon (NFD) that carries out forwarding functions: forwarding Interest packets originating from consumers to their producers and sending back responding Data packets in reversed paths. An Interest packet does not have sender information; thus the network is responsible for recording the packet traveling path. The forwarding plane of NDN is, therefore, stateful since the Interest packet leaves traces at passing-by NFDs like breadcrumbs, for which the returning packet can travel back to its requester. At an NFD, a face is an endpoint where it communicates with local applications or other NFDs at the next hops in the network. The connection between two faces can be implemented with different lower layer protocols such as HTTP, UDP/TCP/IP or Ethernet. The forwarder works as a packet switching machine: when an NDN packet arrives at one face, the engine processes then forwards to another face to send to a next hop.

To carry out the function, an NFD must maintain three data structures: A Pending Interest Table (PIT), a Content Store (CS), and a Forwarding Information Base (FIB) and its Forwarding Strategy. The PIT is organized like a table that keeps all Interest packets that arrive and have not been satisfied with corresponding data packets. An entry registers an Interest packet and the faces where it comes from. The CS records Data packets that have been satisfied. A subsequent Interest, therefore, can be responded immediately with a cached data packet bearing the same name. The FIB keeps routing information to help find the network face through which the Interest packet should be forwarded to the next hop. An entry in the FIB registers a name prefix, a list of face identities, and a forwarding strategy. The name prefix is used for matching any arriving Interest packet, while the forwarding strategy defines how faces should be selected among the list to forward a matched packet.

Figure 1 illustrates how packets are processed at an NFD. When an Interest packet arrives, the NFD searches in the CS for a Data packet matching the name of the Interest packet. If a packet is found, it is returned immediately to the incoming face where the Internet packet arrives. On the other hand, the Interest packet is searched against the PIT to find an existing instance that matches to its name. In the case that a pending Interest is found, the PIT updates the list of incoming faces for the packet. Otherwise, the packet name or its forwarding hint is searched against the forwarding base table with prefix-matching to find the best next-hop to forward. The packet is dropped if no such hop is found, otherwise, it is sent to the face and the PIT adds it to its table.

In the reversed path, when a Data packet arrives, the NFD first looks for the existence of a matched Interest from the PIT. It ignores the Data packet if any matching is not found. Otherwise, the packet is cached at the CS before being forwarded to all the faces which were registered with the corresponding Interest as entries in the PIT.

The implementation of a stateful forwarding plane at NFDs bring several crucial features of NDN:

- *request aggregation*: Not all Interest packets having a same name reach its producer. Instead, they stop at the first NFDs whose PIT already records a packet of the same name except for only one that arrives at the producer. In other words, the Interest packets of the same name are aggregated at NFDs.

- *in-network caching*: Data packets are cached for a certain period at NFDs wherever they travels through. This in-network caching feature allows subsequent requests for a same data can be satisfied immediately at the first NFD having its cached instance.

- *native multicasting*: The paths left by Interest packets carrying a same name from consumers to a producer build up a tree structure: the root is the producer, the leaves are the consumers and the inner nodes are NFDs having the packets. When the producer responds with a Data packet, it travels from the root of the tree to all the leave nodes in a multicasting fashion.

NDN routing scalability

FIB entries can be populated either manually or automatically with an NDN routing protocol [12]. While address space of an IP network has an upper bound, the number of entries in an NFD's FIB can grow extremely large since the name prefix may be significantly longer. This can be a serious scalability issue for NDN deployment. The NDNS [13] service was proposed to overcome this problem. In this approach, the NDN namespace is divided into two groups: the first group consists of name prefixes known by NFDs of the core network while the second group consists of names known only by NFDs at local domains. An Interest packet whose name is unknown to the core network must set its `ForwardingHint` value as the domain name prefix of the producer. The NDNS service provides a mapping between the name of data and the name of the domain where the data is produced. Consumers must query the NDNS service for the domain name of its producer.

3.2. Blockchain technology

3.2.1. Blockchain framework

Most of the current blockchain systems follow a common framework that was introduced in Bitcoin and Ethereum. In general, it can be structured in four layers as shown in Figure 2.

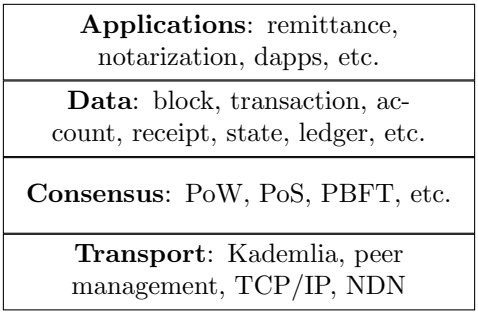


Figure 2: A generic blockchain framework

- *Application layer* comprises a set of blockchain-enabled applications. The use cases of Bitcoin and many others of its variant were mainly limited to cryptocurrency transferring. The cryptocurrency is considered as a store-of-value digital asset and their networks are working as public financial systems. With the introduction of the smart-contract concept, this layer is enlarged to include many decentralized applications such as notarization services, identity management, supply chain management, etc.

- *Data layer* defines both data structures for storing and efficiently manipulating the ledger at a node and complimentary data structures for facilitating consensus and data propagation in the network. In general, the ledger is a chain of blocks where a block links to its precedent by a hash pointer. However, the block structure may be different from one to another in many blockchain systems. In Section 3.2.2, we describe main data structures of the Ethereum blockchain.

- *Consensus layer* defines a protocol for which blockchain nodes follow to build the shared ledger. Each node has a copied version of the shared ledger. Maintaining consistency among all versions would require the nodes to agree on the order of transactions to write. For accelerating the speed of transaction recording, transactions are packed in blocks and the agreement is reached on the order of blocks instead. Many current blockchain systems adopt the PoW consensus. The protocol is known for several limitations including extremely slow transaction processing speed and not having transactional finality due to ledger forking. There are a growing number of research works to develop new consensus protocols to overcome these limitations for supporting many potential use cases of blockchain technology. For a comprehensive review of state-of-the-art consensus protocols, please refer to Du et al. [14].

- *Transport layer* specifies how the blockchain network is constructed and how to propagate blockchain data in the process of recording transactions to the ledger. Public blockchain systems such as Bitcoin or Ethereum operate on a P2P overlay, thus it is scalable and open for participating.

Blockchain data item such such as a block or a transaction is disseminated from a single source to all nodes through the P2P overlay. The Section 3.2.3 describes the transport layer at Ethereum blockchain in more detail.

3.2.2. Ledger data structure

Figure 3 presents a simplified model of the generic structure of a ledger. It consists of a sequence of blocks $\{B_n\}$. The first block B_0 is called genesis block which is predefined from the inception of a blockchain network. A block B_n is made of a block header h_n and a block body b_n. The block body includes a set of transactions T_n in certain order decided by its block creator. The first transaction of the block is for rewarding the creator with a certain amount of cryptocurrency (in cases where an incentive-based consensus protocol is used). A full description of block structure and transaction structure in Ethereum blockchain can be found at Wood et al. [15].

In general, a block header h_n must contain at least three components: the hash of block header h_{n-1}; a root hash value of the Merkle tree created from T_n; and a proof to certify that the block was properly created following the chain consensus rule. In PoW blockchain, the proof is an Integer number.

When a blockchain node receives a block header, it can verify that the header is valid given the following condition are met: the header structure is conformed to the blockchain specification; the header links to a block header (parent block) in its local ledger; and the consensus proof is correct.

When a blockchain receives a block, it can verify that the block is valid given the following conditions are met: the block header is correct; all data integrity checks are passed, that is, the block body content is matched to its fingerprints on the header; and all transactions are valid, that is, they can be executed against the node's local ledger state without any failure.

The integrity of the transaction list is secured by the Merle tree root in its header. The integrity of the ledger is secured by the way blocks are linking one to another with hash pointers. The ledger is, therefore, immutable thanks to this way of structuring because a tiny modification on a transaction can break the integrity of the ledger. In addition, the ledger is self-verifiable, i.e., a node can verify the validity of the downloaded ledger with high certainty without relying on other parties by verifying the proof of every block. For example, in case where a PoW consensus is used, a correct proof indicates that a certain amount of computation has been spent on its block creation. Estimating the total cost of ledger building would let the node trust that no adversary has enough computation capacity and incentive to create such a ledger.

3.2.3. Ethereum transport layer

The transport layer of a blockchain system is responsible for delivering necessary data to blockchain nodes for ledger synchronization. Ethereum uses a gossip-like protocol to broadcast transactions and blocks on its P2P overlay. Realizing this needs to implement three functional components: peer discovery, peer management, and data propagation.

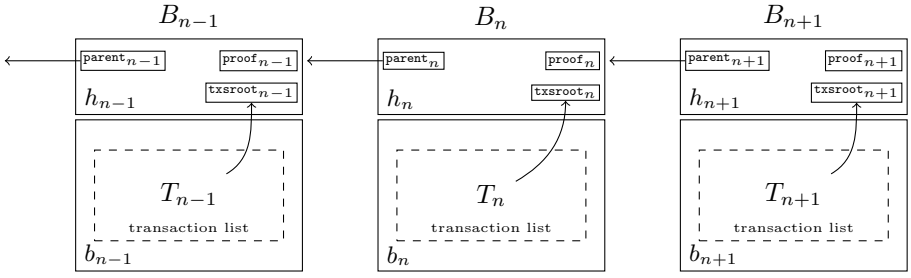


Figure 3: Blockchain data structure

Peer discovery

A public blockchain system operates on a P2P overlay for scalability. It provides the system resiliency against churning, a condition where nodes may come and go without notification. The peer discovery function allows a newcomer to discover existing nodes in the network. To carry out this function, Ethereum implements Kademlia [16] distributed hash table protocol for creating a structured P2P overlay. Nodes in the network gradually build their partial view of the P2P logical topology which helps them reach each others effectively. All messages of the Kademlia protocol are sent with UDP packets. Section 10.2 gives a brief explanation about the protocol.

Peer management

The component allows a node to monitor status of some other peers that it trusts for the purpose of ledger synchronization. These peers are selected from a list of known peers in the past (that was saved from some previous runs) or they can also be selected from the routing table of the peer discovery module. A TCP connection should be established between the node and any selected peer. The connection can be initiated by one of the partner without a priority. The two nodes should exchange a pair of handshake messages in order to assure that they are maintaining the same ledger and both are ready to accept the connection for the purpose of later data communication. If the procedure succeeds, the peer is trusted and its status is continuously updated until one of them abandon the connection. At the end, the node should have a list of trusted peers which it frequently exchanges messages to build ledger.

Data propagation

Blockchain nodes exchange messages to broadcast transactions and blocks. New transactions from users need to be delivered to miners for committing to blocks. As there is no mechanism to differentiate a miner from a non-miner, the transactions are, therefore, broadcasted. Newly mined blocks are also broadcasted in the network for the nodes to update their local ledgers. A node only communicates with the peers in its trusted list for this purpose. Any piece of data for broadcasting is propagated hop-bye-hop in a gossip-like fashion.

A transaction enters the blockchain network at one of the blockchain nodes

from a node user's submission. If the transaction is valid and it has not been included in any existing blocks of the ledger, the node forwards the transaction to all of its trusted peers. When these nodes receive the transaction, they follow the same procedure to carry it further in the network. If the transaction is invalid or it has already been included in the node's ledger, it is ignored and the forwarding does not occur.

Miners of the blockchain system collect transactions to create new blocks. When a new block is mined, its miner immediately announces to the network in order to have it accepted by other nodes for claiming a reward. The block propagation works in the same manner as that of transaction propagation, though with a little tweak. When a node receives a new valid block, it may choose to push the block to only a few peers while sending the block hash to the remaining ones so that they can retrieve the block later. The change is to reduce the amount of unnecessary traffic as a block may occupy much more traffic than a transaction. But on the other hand, fast block delivering is critical for the security of PoW blockchain since higher latency means higher probability of chain fork [17]. Therefore, the number of peers that receive block push is a parameter of the blockchain client which is chosen as a trade-off between block propagation latency and traffic redundancy. A large value would help flood blocks to the whole network quickly but resulting in a significant waste of traffic usage.

The gossip protocol has two limitations: first, the same data item is sent multiple times on different TCP connections which causes unnecessary traffic; second, a non-optimal route is taken for data delivering as nodes are not aware of the physical network topology. These two issues are intrinsic to the P2P content delivery in IP networks. The first problem is tolerable in transactions delivery as their sizes are small, but it is not so in the case of block delivery. The tweak in the block propagation protocol can lighten the issue, albeit with a cost: some block deliveries would require 1.5 round trip message exchange. Since the receiving peer is not aware of the nearest source of the block, it always requests the block from the first announcer, thus the delivery latency is far from being optimal.

4. Design of data propagation protocols

4.1. System model and problem definition

The target blockchain system is running on an ICN network. Specifically, we assume that the NDN realization of ICN is implemented. The system is a network consisting of public computing nodes, each runs a blockchain client application. The clients communicate to exchange messages for synchronizing on a shared ledger. The ledger structure is formally presented by the Ethereum specification [15] (a brief overview is presented in Section 3.2.2). The system uses PoW consensus protocol for adding blocks to the ledger (see Section 10.1). However, instead of using IP-based transport to propagate data for ledger synchronization, the nodes exchange messages in NDN packets.

The system users utilize blockchain network to facilitate their application use cases. Typically, users send transactions to the system for recording to

the shared ledger. In order to carry out the request, a user may run its own blockchain client or it has a trusted agent which is running a client to act on its behalf. The user sends transactions to the system through the client. It also may query the client for information on existing transactions. The interface between the user and the client program, however, is not discussed in this work. If a transaction is valid, the blockchain system should record the transaction to the shared ledger.

When a transaction is sent onto the system, the nodes communicate to propagate the transaction to the whole network so that eventually it can be included in a valid block on their local ledgers. The nodes follow PoW consensus protocols to decide on the next block. There are some mining nodes that invest their computing power to create blocks for receiving rewards. They collect transactions on the network to create blocks and try to solve the PoW puzzle on the blocks. Once a new block is successfully mined, its miner sends the block to the network. Again, the blockchain network delivers the new block to the nodes. When receiving the block, a node includes the block in its local ledger given that it can verify the content of the block. Eventually, all the local ledgers of the nodes converge to a single consistent ledger.

In this work we implement a blockchain client to realize a blockchain system on NDN networks. In order to realize that, we design a new transport layer for delivery of blockchain data over NDN infrastructure. We assume that the client application is running on a host having a local NFD that connects to the NFD of a router on the NDN network. The client, therefore, interacts with the NDN networks by sending and receiving NDN packets through the local NFD. The node has a name that we called *nodename* which is given by its administrator with an out-of-band method. We assume the number of names is limited for a given a node. When the local NFD starts, it should register the name to the connecting NFD. The registration allows the network to direct any subsequent Interest packets whose forwarding information includes *nodename* as a prefix toward the local NFD. The forwarding information can either be the packet name or the `ForwardingHint` element.

Throughout the rest of the paper, we use `/ethchain` as the NDN application name of the client for composing NDN packets. Let `/etri/bob` be the *nodename* of an example host, then the network and the local NFD are responsible for directing any Interest packets whose forwarding information includes `/etri/bob/ethchain` to the client.

We assume that client p of blockchain network posses a pair of cryptographic keys pub_p and prv_p. We denote $sig_p \leftarrow \texttt{Sign}(d, prv_p)$ as a signing function that generates a signature sig_p for data d with a private key prv_p; `true`/`false` $\leftarrow \texttt{Verify}(sig_p, d, pub_p)$ as boolean function that verifies the correctness of the signature sig_p signed on data d given the public key pub_p; $h \leftarrow \texttt{Hash}(d)$ as a cryptographic hash function that returns a hash string h of the input data d. A generic data structure obj is encoded/decoded to/from a binary byte array $wire$ by method `BinEncode`(obj)/`BinDecode`($wire$).

When describing the protocols, we describe a message in the format $\langle MsgType, attr1, ...attr_n \rangle$ where the first element is message type and the

subsequent elements are content attributes of the message. We denote msg_{δ_p} as a signed message created by node p using its private key. It is composed by attaching its original version with a signature as $msg_{\delta_p} \leftarrow \langle msg, \texttt{Sign}(\texttt{BinEncode}(msg), prv_p) \rangle$. A receiver should drop a signed message if it can not verify its signature.

4.2. System requirements

A naive implementation of blockchain on NDN would create a TCP-like connection between nodes using NDN Interest/Data packets and then reuse current Ethereum data propagation protocols on those connections. However, the approach has all the limitations of the IP-based blockchain. The new protocols should take advantage of the NDN transport to minimize the traffic redundancy in data delivery without sacrificing the security of the blockchain network.

The target system is a public blockchain, thus anyone is free to join the network without requiring special policy from its network operators. We suppose that any ordinary node which is granted a prefix by its network administrator (so that it can be a data producer) can participate in the blockchain system without any further requirements. In addition, we require that our design should respect the scalability of current public blockchain systems.

The system must be compatible with the existing implementation of the Ethereum blockchain. That is, we must keep the storage layer of the Ethereum blockchain intact. A valid transaction on IP-based Ethereum blockchain should be valid on the NDN-based blockchain. As a result, decentralized applications using smart contracts can be deployed on both blockchain systems without modification.

Cache poisoning attack is difficult to deal with in NDN [18]. Since the blockchain is a public system, its working environment is expected to be hostile. Thus we should expect the attack is prevalent. An adversary may create fake data to prevent honest nodes from retrieving needed data which can be a starting point for mounting other attacks. The designed system must, therefore, have the capability to overcome such kind of malicious acts. More specifically, a poisoning attack may hinder the service performance of the system but no functional interruption is allowed to happen.

4.3. Design considerations

Although ICN has become mainstream in network research, it is still in the early stage. There is a lack of common knowledge on how a real system is deployed and what kinds of assumptions should be taken before designing new distributed applications. As a result, it is usually the case that some limited features of ICN are employed exclusively for application protocol design. For example, a frequent pattern of designing distributed systems [19], [20], [21], [22], [9], [11] is the use of a common routable prefix for naming data items and assuming that any participating node can produce data content with it. The validity of this assumption is susceptible since its acceptance also means that all the network routers must have the prefix in their forwarding table.

This condition may be satisfied in limited settings, but it is not applicable in generic cases, especially for scalable public systems. Such an assumption would jeopardize the scalability of the network deployment [13] since there is an upper limit on the capacity of forwarding table at every routers. In our design, we should avoid to make use of any restricted feature of ICN.

As multicasting is natively supported by NDN, it is tempted to consider abandoning the P2P overlay for the content delivery in blockchain systems. This is the approach adopted by some earlier works [9],[11]. For example, as soon as a new block is inserted, blockchain nodes can anticipate the identity of the next new block by its block number. They simply request it by sending NDN Interest packets that share a same name with the block number as one of its name parts. The name should have a predefined common prefix which is known by all the nodes. All the miners in the network who can be potential block creators must be given the right to produce data on that prefix. When one of the miners wins a block, it can satisfy the requests with the block data. The block (or its segments) then be multicasted natively from its creator to all blockchain nodes efficiently.

This approach has two serious problems: first, it leads to data conflicts as miners can produce blocks at the same time; second, it may not feasible in a public system. It may work in a private system where participating nodes are trusting each other. In a public system, it is prone to poisoning attack as a malicious player can easily satisfy the requests with an invalid block causing unnecessary traffic and computation cost to the system. Therefore, we should not use the anticipating guess of new data for making requests. Instead, the availability of the data must be announced explicitly.

If new data must be announced before it can be retrieved, are there any NDN supported methods for broadcasting the announcement? Indeed, NDN has such a mechanism and it is exploited by Chronosync and its variants. Announcement messages are named with a prefix that receives a special treatment from NDN routers. The routers must enable the multicast forwarding strategy for a given prefix. That is, multiple producers can register for the same prefix and routers forward arriving Interest packets that match to the prefix to all the registered producers. It still remains doubtful if such treatment is admissible for the deployment of public services such as blockchain systems.

In addition, the broadcasting announcements using Interest packets is not useful for blockchain data delivery and in any general public system as it is open for harmful acts. For example, if transactions can be broadcasted instantly to all nodes, a malicious node can flood the network with invalids transactions to cause the system to spend computational resources for processing them. In current blockchain systems, the P2P overlay plays an important role in preventing such kind of attack. Honest nodes only forward valid transactions in the gossip delivery blocking any attempts to flood invalid transactions to the network. Therefore, we believe that the P2P overlay is indispensable for building a scalable blockchain system over NDN.

4.4. P2P overlay over NDN

In this section we describe how Kademlia-based peer discovery is implemented in NDN and how a node establishes and manages a list of peer connections for blockchain data delivery.

4.4.1. Peer discovery

Ethereum uses Kademlia, a popular Distributed Hash Table (DHT) protocol for building a structured P2P network. Shifting its underlying transport from IP to NDN should not change its main logic, thus we do not discuss its implementation. However, the RPC primitives must be redesigned to work on NDN networks. This can be done easily with Interest/Data receiver-driven communication model. A client can embed its request in an Interest; a server responds with the RPC call results in a returning Data packet. An RPC request/response message is small enough to be embedded in a single NDN packet.

Node record and node identity:

A node in Kademlia P2P networks has a globally unique identity. It is a fixed-length random bit string generated by hashing a public key and auxiliary data. Let prv_p and pub_p is a pair of a private key and a public key owned by node p and let $nodename_p$ be its NDN *nodename*; We define node record of p is a binding of its public key and its prefix as $rec_p \leftarrow \langle pub_p, nodename_p \rangle$. Node identity of p is the cryptographic hash value of the binary encoded node record: $id_p \leftarrow \texttt{Hash}(\texttt{BinEncode}(rec_p))$. We use Keccak [23] for the hash function and the output identity is a 20-byte bit string.

RPC messages:

RPC message must include a node record identifying its sender and it must be signed by message sender. By including the binding of sender's public key and its *nodename* as well as requiring the message to be signed, the system can prevent Sybil attack as adversary is restricted on the number of node names he can own.

An RPC request message is represented as $\langle \texttt{RPC\_REQUEST}, rec_p, m, argv, n \rangle_{\delta_p}$ where m is the name of RPC method which can either be `Findnode` or `Ping`; $argv$ is the list of arguments used by the method; n is a random nonce value that uniquely identify the RPC call. The random nonce inclusion guarantees that response to a request is not received from cache. An RPC response message is represented as $\langle \texttt{RPC\_RESPONSE}, rec_p, rst, err \rangle_{\delta_p}$, where rec_p is the node record; rst and err represent result and error of the RPC call, respectively. The response message does not include the *nonce* of its request because matching is implicated in the Interest-Data packet name-matching.

4.4.2. Peer management

A node maintains a list of trusted peers with which it keeps communicating to exchange data for blockchain synchronization. A peer can be trusted if it has the same ledger. Whenever a peer is suspected of acting maliciously, it can be dropped from the trusted list and it may be added to a blacklist for preventing

it from re-connecting in a certain period. Each peer in the trusted list must have an attached node record which consists of its public key and its NDN node name. The peer identity can be infer from the node record as described in Section 4.4.1. The node can query for a peer from the trusted list given the peer's identity or it's node name.

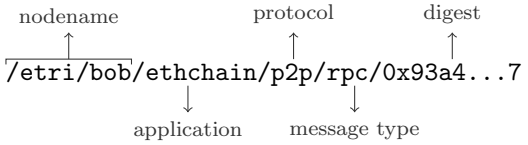
The peer management protocol uses two types of signed messages: $\langle \texttt{STATUS}, rec_p, accept, chainstatus \rangle_{\delta_p}$ and $\langle \texttt{BYE}, rec_p \rangle_{\delta_p}$. The former is used to exchange chain status (*chainstatus*) information for peer handshaking, the latter is used for notifying of peer leaving. The messages must include a node record for identifying sender. The $chainstatus_p$ should include information needed for deciding if the peer owning rec_p can be trusted. The *accept* parameter indicates handshake acceptance status. If its enclosed message is for handshake offering, its value is always `true`.

A `STATUS` message can be encapsulated in either an Interest packet or a Data packet depending on who is starting the handshaking. A `BYE` message is encapsulated in an Interest message when a node wants to notify the receiver that it has been dropped. It can also be encapsulated in a Data packet in responding to any unsolicited Interest packets, which means that the requesting peer is not in the trusted list.

As our peer management is similar to that of IP-based Ethereum client except for the fact that the protocol messages are delivered on NDN, we refer readers to Section 10.3 for its detailed description.

4.4.3. NDN packet encapsulation and naming

When encapsulating a P2P message in an NDN Interest, we set the `AppParams` element of the packet as the binary data of the message. The packet name is then composed by concatenating the *nodename* of the message receiver, the application name, the protocol name, the message type, and the hex string of a digest. The message type component can be `/rpc` for a `RPC_REQUEST`, `/status` for a `STATUS` message, and `/bye` for a `BYTE` message. The digest is the hash value of the `AppParams` element and it is estimated as `Hash`(`BinEncode`(msg)), where msg is the message to be encapsulated. Below figure shows a naming example to illustrate the scheme.



A response message msg_{δ_p} is encapsulated into a Data packet by setting the `Content` element of the packet as `BinEncode`(msg) while its embedded signature is used to set the value of the `Signature` element of the packet.

4.5. Data propagation protocol

4.5.1. Announce-pull data propagation scheme

In this section, we present the announce-pull data propagation scheme that will be applied to both block delivery and transaction delivery. The approach can take full benefit of in-network caching and native multicasting features of NDN so that sending redundant traffic can be avoided. The core idea is to use a gossip-like protocol on P2P overlay to broadcast the announcement of new data. Once the peers know of the announcement, they can pull the data from the network using a unique name that can be inferred from the announcement. Uniquely naming the data allows its requests to be aggregated and the returning data packet can be cached by the network and efficiently multicasted to all requesters.

To realize the idea, an announcement must carry two pieces of information: identity of the announced data item and its location. The former is necessary for creating a unique name for the item; a hash value of the data can serve this goal. The later is needed by the network to forward a data request to its producer. Since an Interest packet has no source information, an announcer need to attach its node identity to the announcement for an receiver to determine its location. The location information is the *nodename* of the announcer which the receiver can find by querying its trusted peer list for the announcer identity. When pulling the data, the receiver puts this information into the `ForwardingHint` element of an Interest packet. All in all, the P2P overlay is acting as a dynamic name resolution system that maps a data item to its location (*nodename*).

The example shown in Figure 4 illustrates the idea. The protocol takes place at a blockchain node after it has discovered other peers in the network and has established a list of trusted peers to exchange blockchain data. Let's suppose that the peer p_0 has a new data item that needs to broadcast to the network. It can be a new transaction or a newly mined block. The peers p_1 and p_2 are in its trusted list. The peer p_0 encapsulates an announcement of the new data into Interest packets and sends them to the other peers individually. These packets bear different names although they have the same announcement data. The announcement should have information that uniquely identifies the data item and the identity of p_0. The peers p_1 and p_2 extract the announcement, and then create Interest packets to request the data. These packets have an identical name `/ethchain/7a...e` composed of a common predefined prefix and a hash string, which is calculated from the announcement information. Since the name has no routable information, the packets must set the `ForwardingHint` element as the *nodename* of p_0 so that the network can forward the packet to the data producer. p_1 and p_2 can infer the *nodename* of p_0 by querying their trusted peer lists using the identity of p_0 as input.

Having the same name allows the aggregation of the packets along the paths to the producer. At the end, only one of them arrives at the destination. The producer p_0 then packs the data item in a Data packet with the same name, and then sends it to the network. The packet then travels in reversed paths to the requesters p_1 and p_2. If the peers p_1 and p_2 can validate the received data

item, they will announce it to the peers in their trusted list. On the other hand, if p_0 is sending an invalid data item, p_1 and p_2 will discard the data preventing its propagation.

Let p_3 be one of the peer that receives a subsequent announcement from p_2. It will request the data item with an Interest packet having the same name as previous ones sent by p_1 and p_2. However, the `ForwardingHint` element of the packet should be set as the prefix of p_2. The request will not reach p_2 because it is aggregated at an NFD along the path to p_2. The peer will receive a cached version of the data item returned by that same NFD.

Dealing with cache poisoning attack

Since only announcements with validated data are forwarded, an invalid announcement can only travel for a few hops before it is completely blocked by the honest nodes of the system. If a node sends an invalid announcement, eventually it will be requested to returned a data item for the announcement by all receivers. If the node does not response or it returns with an invalid one, all receivers will drop it from their list of managed peers and prevent it from re-connecting in a certain period. Therefore, we should have no worry of malicious nodes sending invalid announcements.

However, our protocol has to deal with the case where a valid data item sent by an honest producer can be prevented from multicasting to honest nodes. In a normal situation where there are no attempt of cache poisoning, the data delivery should be efficient as designed. However, an adversary owning multiple malicious nodes can sabotage the process by mounting a poisoning attack. As soon as one of its nodes receives an announcement for a valid data item, it starts requesting the item from another malicious node who returns a faked item. Such a strategy would trick the network to cache the faked item for the name which should be reserved for the valid one. As a result, a subset of NFDs may have a poisoned cache which causes the other honest nodes to waste their effort on requesting the wrong data.

Such an attack can not be prevented instantly as the NFDs do not check for data integrity. However, the attack should not prevent the delivery of the data item to the honest nodes. This can be done by requesting the data item directly from its announcer. When receiving invalid data for a given announcement, an honest node makes other iterative attempts to fetch the data. But these times the node uses the prefix name of one of its announcers when composing the name of the request packet. If that chosen announcer return an invalid data item, it is immediately added to a blacklist preventing it from future attacks. The node may try multiple times until it retrieves the expected data item from another honest node.

4.5.2. Protocol messages

We apply the announce-pull data propagation scheme for the delivery of transactions, block headers and block bodies. A block can be reassembleb from its header and its body. The following messages are used in our protocols:

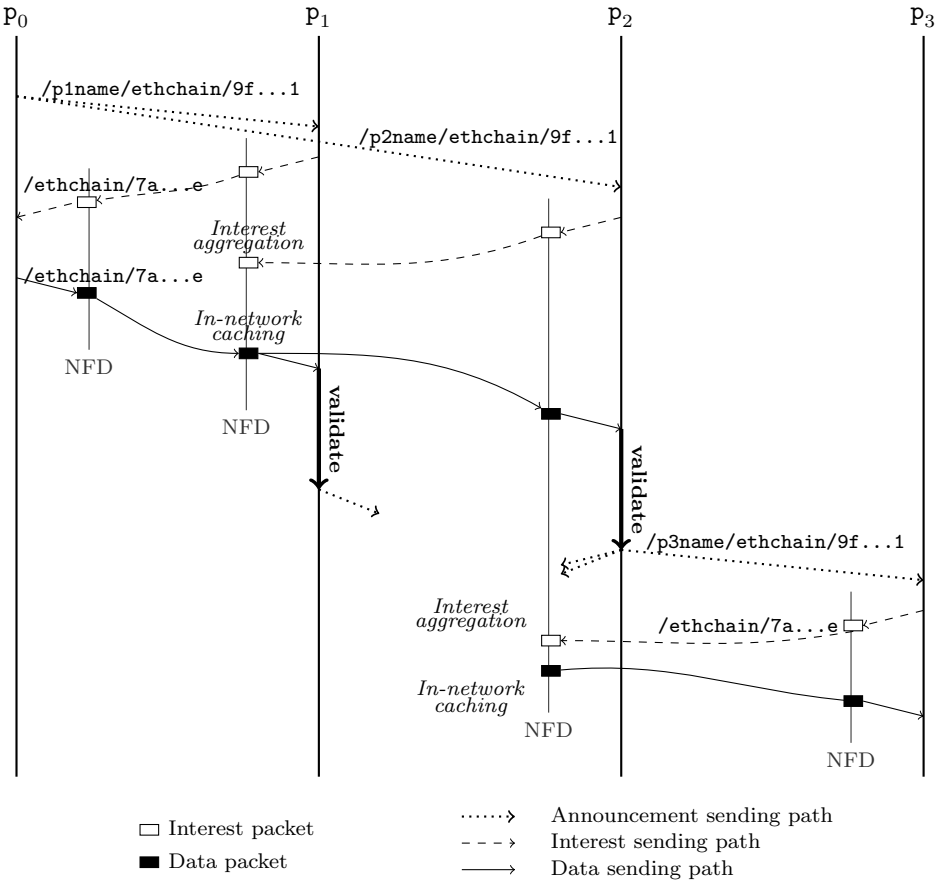


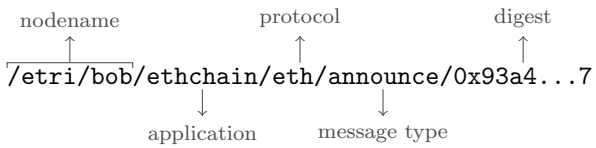
Figure 4: Announce-pull data propagation scheme

- $\langle$`ANNOUNCEMENT`$, id_p, annlist\rangle_{\delta_p}$: an announcement message notifies the existence of data items which are available for retrieving. The message includes the peer identity id_p, and a list of announcements *annlist*. The message must be signed by its sender and its receiver must verify the signature before processing.
- $\langle$`DATAREQ`$, type, hash, segid\rangle$: a data request message is used to fetch a segment of data. The data item can either be a transaction, a block header, or a block body indicated by *type*. The message contains a hash value *hash* which can be used to query the data item, and a segment identification *segid* to indicate which segment to be retrieved.
- $\langle$`DATASEG`$, wire, num\rangle$: a data segment message is a response to a `DATAREQ` message. It contains the byte array *wire* representing the binary array of the requested segment and the total number of segments of the requested item.

An announcement is a tuple $\langle type, content, num\rangle$, where *type* indicates whether the announced item is a transaction, a block or a block header, *content* is the content of the announcement, and *num* is the number of segments of to-be-requested item. The meaning of the last two parameters depends on the context set by the announcement type. In case the announced item is a transaction, the *type* element is set to `ANN_TRANS`, the *content* is the hash value of the transaction and the *num* indicates the number of segments of the transaction in binary. If the announced item is a block, the *type* element is `ANN_BLOCK`, the *content* element is set as the hash of the block, the *num* value indicates the number of segments of the block body in binary. If the announced item is a block header, the *content* is a signed tuple $\langle header, rec_s\rangle_{\delta_s}$ where *header* is the block header and rec_s is the node record of the block owner s, the *num* value indicates the number of segments of the block body in binary. It is worth to note that an announcement of type `ANN_HEADER` does not indicate that the announcer already has the block. Instead, it indicates that the announcer only has the block header and the block can be fetched from the peer s who signs the announcement.

4.5.3. NDN packet naming

`ANNOUNCEMENT` message is encapsulated in an NDN Interest packet for sending directly to a peer. We apply the same method to build the packet as for P2P messages. The binary content of the message is used to set the `AppParams` element. The name is a concatenation of the receiver host name, the application name, the protocol name, and the digest of the message binary in hex string. Below is an example illustrating the naming method.

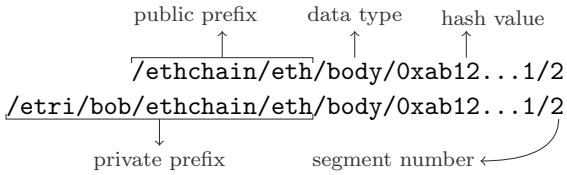


When a node receives an Interest packet with an `ANNOUNCEMENT` message, it should response with a Data packet. The packet may have no content. However, when the node has data to announce, it may inject an `ANNOUNCEMENT` message in the Data packet by setting the `Content` element as the binary of the message content and the `Signature` element as the signature generated on the former. For the shake of simplicity in later description of protocols, we assume that the Data packet has empty content. But in practice, `ANNOUNCEMENT` messages are delivered using both methods, and they are treated identically when received.

A `DATAREQ` message is used to request a segment of data item and it is encapsulated in an NDN Interest packet. We use the content of the message to build the name of the Interest packet as follows.

To name the Interest packet, a node may use two kinds of prefixes: a private prefix to retrieve data directly from its producer; and a public prefix to retrieve data from the network without specifying the data producer. A public prefix is commonly known by all nodes. We set it as `/ethchain/eth` which is composed of the application name (`/ethchain`) and the protocol name (`/eth`). The prefix is not routable globally; that is, the routers do not keep it in their RIB tables. When sending an Interest packet using the public prefix in naming, the packet must include forwarding information in the `ForwardingHint` element to help routers know where to direct the packet. A private prefix is created by concatenating a *nodename*, the application name and the protocol name. It should be routable globally because it begins with a *nodename*. When trying to retrieve a data item, a node should names the requesting Interest packet using the public prefix. If the first attempt fails as the received packet is polluted, the node should request the data from its announcers using private prefixes.

The name of an Interest packet enclosing a `DATAREQ` message is composed by concatenating one of the prefixes, the requested data type, a hash string representing data identity, and a segment number. The name component representing a data type can be either `/tx` for transaction, `/header` for block header, and `/body` for block body. For example, to request for the segment number 2 of a block body with block hash `0xab12...1` from peer `/etri/bob`, the name of the request packet can be set as one of the following names:



A `DATASEG` message is used to response to a `DATAREQ` message, thus it is encapsulated in an NDN Data packet. Given a response message msg, we use its binary `BinEncode`(msg) to set the `Content` element and its hash value `Hash`(`BinEncode`(msg)) to set the `Signature` element of the packet.

To simplify the explanation in the later sections, we define two procedures: `SendInterest`($msg, prefix, usehint$) and `SendData`(msg). The first procedure

implements the sending of an Interest packet that encapsulates a protocol message *msg*. The boolean argument *usehint* indicates if a public prefix (`true`) or a private prefix (`false`) is used for naming the packet. The $prefix$ is the *nodename* of the producer. In case where a public prefix is used in naming, the $prefix$ is used to set the `ForwardingHint` element of the packet. Otherwise, it is used to create a private prefix to compose the name of the packet. The second procedure implements the sending of a Data packet in response to a received Interest packet (which is implicitly known by the method).

4.5.4. Sending announcements

Algorithm 1 Sending announcements at node $p^\star$

```
1:  Variables
2:      id_{p*}: node identity
3:      prv_{p*}: private key of p*
4:  procedure SendAnnouncement(annlist, peers)
5:      for p is in peers do
6:          sendlist ← remove known announcements by p from annlist
7:          msg ← ⟨ANNOUNCEMENT, id_{p*}, sendlist⟩_{δ_{p*}}
8:          SendInterest(msg, p.Prefix(), false)
9:          update p's known hash list with sendlist
10:     end for
11: end procedure
```

In this section, we describe how sending announcements is implemented for a blockchain node. The node should not send duplicated announcements to a peer, and he should not return a received announcement because it wastes computation resource and incurs unnecessary traffic. Therefore, the node should maintains a list of data items known by each member in its trusted peer list. We use the hash of a data item (either transaction or block) as its identity to manage the list. The list gets updated whenever the node sends/receives an announcement to/from the peer. Before sending a new announcement, it should assure that the hash value embedded in the announcement is not known by the peer.

We define a procedure `SendAnnouncement` on the Algorithm 1 that implements the sending announcements. The procedure is used exclusively to implement the transaction propagation and block propagation. The input arguments are a list of announcements *annlist* and a list of receiving peers *peers*. The main body of the procedure loops over the list *peers* to execute following steps on a peer p.

- line 6 - a new announcement list *sendlist* is generated by filtering from *annlist* any announcements whose embedded hash value has been known by p.

- line 7 - an announcement message is created with *sendlist* and signed by private key $prv_{p^\star}$.
- line 8 - the message is sent to peer p by calling the method `SendInterest`
- line 9 - embedded hash values in *sendlist* are added to the list of known hash values of p.

4.5.5. Data serving and fetching

A blockchain node acts as a producer and a consumer at the same time. It needs to request data from other peers, but also it can serve data to them. Fetching and serving data are the building blocks for blockchain data delivery. In contrary to the data sending on TCP streams, data segmentation must be handled by applications. Fetching data items may require multiple rounds of exchanging the message pair `DATAREQ`/`DATASEG` encapsulated in NDN packets. Algorithm 5 in Section 10.4 sketches the procedures for serving and fetching a generic data item in our blockchain implementation. We define a method `Fetch`$(type, hash, num, prefix)$ which implements the fetching from network a data item of type *type* having *num* segments and being identified by *hash*. The parameter $prefix$ is the node name of the data's owner. The argument *type* can be one of three values: `TRANSACTION` for fetching a transaction, `HEADER` for fetching a block header, or `BODY` for fetching a block body. The method is used exclusively in the transaction propagation and block propagation algorithm.

4.5.6. Transaction propagation

A blockchain node maintains a transaction pool (*txpool*) to keep newly arrived transactions. They are collected for mining new blocks. The transactions can arrive at the node either from other connected peers in its trusted list *peerlist* or they are submitted locally by the node's users. Only valid transactions (with correct signature and format) are kept in the pool. When a newly mined block is inserted into the ledger, all of its transactions are excluded from the pool.

Nodes in the blockchain system exchange messages to propagate transactions that are newly added to their transaction pools. Algorithm 2 implements the transaction delivery protocol at a node.

(Lines 4-11): the handling function starting at line 4 is triggered when a set of new transactions *txs* is added to the transaction pool. For each transaction *tx* from the list, a transaction announcement is created with its hash value and the number of segments *num* of its binary content to build an announcement list *annlist*. The method `SendAnnouncement` is then called to multicast *annlist* to all the peers in *peerlist*.

(Lines 12-20): this part implements the method to handle the event of receiving a $\langle \texttt{ANNOUNCMENT}, id_p, annlist \rangle_{\delta_p}$ message. The message is processed only if its sender can be identified from *peerlist* and its signature can be verified. For every announcement *ann* of type `ANN_TRANS` in the list *annlist*, its corresponding transaction *tx* is fetched given that it has not been found in the

Algorithm 2 Transaction propagation protocol at node $p^\star$

Variables
    $peerlist$: trusted peer list
    $txpool$: transaction pool
On receiving new transactions txs from $txpool$ **Do**
    $annlist = []$
    for tx in txs **do**
        $h \leftarrow tx.\texttt{Hash}()$
        $num \leftarrow$ calculate number of segments for $\texttt{BinEncode}(tx)$
        $annlist \leftarrow annlist.\texttt{Add}(\langle \texttt{ANN\_TRANS}, h, num \rangle)$
    end for
    $\texttt{SendAnnouncement}(annlist, peerlist)$
On receiving $\langle \texttt{ANNOUNCEMENT}, id_p, annlist \rangle_{\delta_p}$ ▷ an announcement arrives
    get p from $peerlist$ then verify message signature
    for ann in $annlist$ and $ann.type$ is $\texttt{ANN\_TRANS}$ **do**
        $h, num \leftarrow ann.\texttt{Hash}(), ann.num$
        if h not in $txpool$ **then**
            $tx \leftarrow \texttt{Fetch}(\texttt{TRANSACTION}, h, num, p.\texttt{Prefix}())$
            add h to p's known hash list
            add tx to $txpool$
        end if
    end for

txpool. The method `Fetch`(`TRANSACTION`, *ann*.`Hash`(), *ann.num*, *p*.`Prefix`()) is called for fetching the transaction (the method *ann*.`Hash`() retrieves embedded hash value). The transaction is then added to the *txpool*. Its hash value is added to the list of known hash values by the announcer peer p. It is worth mentioning that transaction validation is carried out during its inclusion in the pool. A successful inclusion would then trigger the handling method at line 5 for announcing the transaction to other peers.

4.5.7. Block propagation

Algorithm 3 implements the block delivery protocol at a given peer $p^\star$ following the announce-pull propagation scheme.

When a new block is successfully mined, its miner immediately sends the block to the network to claim the reward. When receiving the block, other honest miners have the incentive to forward the block quickly as their work on the next block has a better chance of acceptance by the network only if the received block is also accepted. It is critical to have blocks broadcast as quickly as possible to reduce the chance of chain forking which in turn can lower the risk of selfish mining and other potential attacks [17]. Therefore, a block should be announced as soon as its header is received and verified instead of waiting for the full block to be fetched and verified. Usually a verified header means that a computational work has been spent on the header; therefore, it is safe to broadcast it without worrying of flooding invalid information.

(Lines 5-9): in case $p^\star$ is mining, the method in this part implements the handling of a newly mined block event. This is where the mining node starts announcing the block to the network. The block header and the number of segments of the block body's binary are retrieved from the newly mined block B. An announcement of type `ANN_HEADER` is created from *header* and *num* value and it is signed with $prv_{p^\star}$. The procedure `SendAnnouncement` is called to announce the header. It is worth mentioning that the miner always pushes a header directly to all of its trusted peers since the size of a block header is small.

(Lines 10-38): the method in this part handles the event of receiving an announcement message $\langle \texttt{ANNOUNCEMENT}, id_p, annlist \rangle_{\delta_p}$.

The message announcer p is identified by querying *peerlist* with identity id_p. Then the message signature is verified with the public key of p. The message is dropped if p can not be found or the signature verification fails.

Each announcement *ann* in the list *annlist* from the message is processed sequentially in a loop. The inner body of the loop contains two main blocks that handle the two cases: a block announcement and a header annoucement.

In case where *ann* is a block announcement (lines 13-22), block header (*header*) and block body (*body*) are retrieved by calling `Fetch` method with input parameters are extracted from *ann*. Note that a block header always fit in a single NDN Data packet; thus, the number of segment is 1. Once the two parts are fetched, the full block B can be assembled. The block header is then verified to exclude the cases it is invalid or it is not suitable to extend the ledger.

Algorithm 3 Block propagation protocol at peer $p^\star$

Variables
  $peerlist$: trusted peer list; $ledger$: the blockchain
  $prv_{p^\star}$: private key of $p^\star$
  $maxpushpeers$: maximum number of peers to push a header
On new block B is successfully mined **Do** ▷ In case $p^\star$ is mining
  $header, num \leftarrow$ get header and number of segments from B
  $ann \leftarrow \langle \texttt{ANN\_HEADER}, header, num \rangle_{\delta_{p^\star}}$
  `SendAnnouncement`$([ann], peerlist)$
  insert B to $ledger$ ▷ add the block to the ledger
On receiving $\langle \texttt{ANNOUNCEMENT}, id_p, annlist \rangle_{\delta_p}$ ▷ An announcement arrives
  get p from $peerlist$ then verify message signature
  for ann in $annlist$ **do**
    if $ann.type ==$ `ANN_BLOCK` **then**
      $prefix \leftarrow p.$`Prefix`$()$
      $header \leftarrow$ `Fetch`(`HEADER`, $ann.$`Hash`$(), 1, prefix)$
      $body \leftarrow$ `Fetch`(`BODY`, $ann.$`Hash`$(), ann.num, prefix)$
      $B \leftarrow$ assemble block from $body$ and $header$
      if $header$ is verified **then**
        $newann \leftarrow \langle \texttt{ANN\_HEADER}, header.\texttt{Hash}(), ann.num \rangle_{\delta_{p^\star}}$
        $peers \leftarrow$ get $maxpushpeers$ peers to announce header
        `SendAnnouncement`$([newann], peers)$
      end if
    else if $ann.type ==$ `ANN_HEADER` **then**
      verify signature of ann
      $header$, $prefix \leftarrow$ get header and block owner's prefix from ann
      if $header$ is verified **then**
        $peers \leftarrow$ get $maxpushpeers$ peers to announce header
        `SendAnnouncement`$([ann], peers)$
      end if
      $body \leftarrow$ `Fetch`(`BODY`, $ann.$`Hash`$(), ann.num, prefix)$
      $B \leftarrow$ assemble a block from $body$ and $header$
    else
      continue ▷ transaction announcement
    end if
    insert B to $ledger$
    $newann \leftarrow \langle \texttt{ANN\_BLOCK}, header.\texttt{Hash}(), ann.num \rangle$
    `SendAnnouncement`$([newann], peerlist)$ ▷ announce to all peers
  end for

If verification succeeds, an header announcement is generated and signed with prv_{p^*}. The announcement is then sent to $maxpushpeers$ number of peers in $peerlist$ by calling `SendAnnouncement` method.

In case where ann is a header announcement (lines 23-31), its embedded header is verified for annoucing immediately. If the verification succeed, the announcement is forwarded to $maxpushpeers$ number of peers from $peerlist$ by calling the method `SendAnnouncement`. Block body ($body$) is then retrieved by calling `Fetch` method with input parameters are extracted from ann. Note that name prefix of the block owner from ann is used for fetching instead of name prefix of the announcer. The full block B is then assembled from $header$ and $body$.

Finally, the full block B is inserted into the node's local ledger (line 35). If the insertion succeeds, the node creates a block announcement of type `ANN_BLOCK` then calls the method `SendAnnouncement` to send it to all peers in $peerlist$. Note that within the method the announcement is checked against the list of known hash values of each peer before sending; thus, no dupplicated sending is guaranteed.

5. Implementation and Evaluation

5.1. Implementation

We use the open-source golang Ethereum client (`geth`) release version 1.8.27 [24] as code base for our project. For NDN packet encoding and decoding, we use the NDN client library that is taken from ndn-dpdk[25] project. Our implementation is planned to be open-sourced and published.

We borrow the general design of `geth` and replace its transport layer part with our design. In general, we remove completely the IP-based P2P layer and replace with our new implementation of NDN-based Kademlia peer discovery and peer management. The block and transaction propagation modules in the original design is replaced with our new implementation to reflect the new protocols. Other parts that are tightly coupled to these components are also replaced or modified to adapt to the changes. We retain the original implementation of main data structures (ledger, block, transaction etc), their serialization/deserialization mechanism, and the consensus protocol. Our blockchain client is, therefore, fully compatible with the original implementation. A ledger created from the Ethereum main-net is usable by our client and vise versa. For a complete description of functional architecture of our design refer to Section 10.5.

5.2. Performance evaluation

In this section we investigate some aspects of our NDN-based blockchain data propagation protocols and their implementation through experiments. The first experiment is about comparing the traffic usage of an NDN-based blockchain system and an IP-based blockchain system in growing a ledger for the same number of transactions. Our expectation is that our protocol can take benefit of the multicasting and in-network caching features from NDN to improve the efficiency of traffic usage in data propagation. The goal of the second experiment

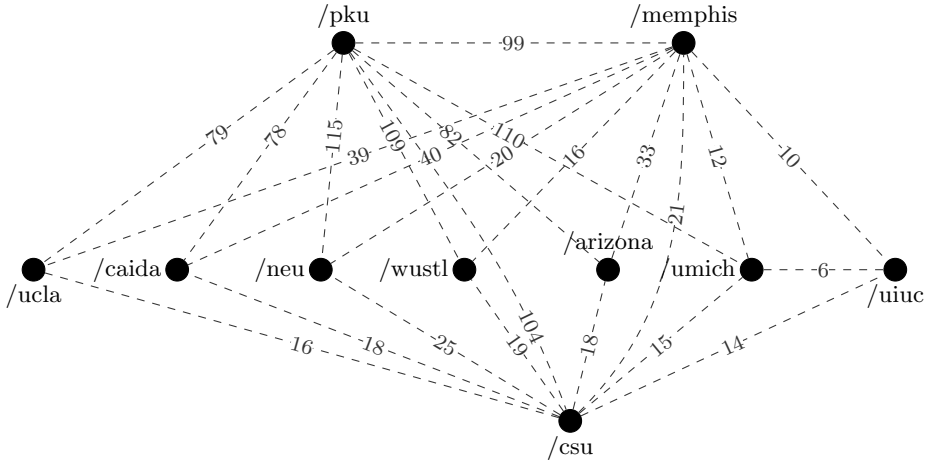


Figure 5: Topology of emulated network

is to measure the block propagation latency, an important measure in PoW blockchain, as we want to investigate efficiency of our protocols comparing to that of IP-based gossip protocols.

5.2.1. Experiment settings

We use `mininet` [26] to emulate a network for conducting the experiments. Due to the limitations of current NFD performance and high computational demand of blockchain clients, it is not possible to emulate a large network on a single physical machine. Therefore, we use the cluster version of the `mininet` to deploy the network on multiple servers.

We conduct our experiments on a local computing cluster. The system consists of 10 servers. Each server is equipped with an Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz, 32 MB of RAM. The servers are running Linux Ubuntu 16.04.1. We install ndn-cxx library version 0.7.1 and NFD version 0.7.1 on every servers. For running IP-based Ethereum blockchain systems, we use golang Ethereum client 1.8.27 which is compiled from source code. This is the same code base that we use to develop the NDN-based Ethereum client.

All of our experiments are conducted on a network topology which is taken from the topology library of `mini-ndn` [27] package. The network consists of 10 routers connecting 10 domains of US universities and institutions. Figure 5 shows the connectivity and link latency among the routers. The routers are equally distributed on the cluster system so that each router is located on a single server. We assume that a single server represents a single network domain. On each experiments we deploy an equal number of blockchain nodes on each domain connecting to the global network through the domain's router.

When emulating in NDN mode, each domain is given an NDN prefix as

shown on Figure 5. The hosts on a domain are enumerated and given names according to their index. For example the first host on domain `/ucla` is given a ndn name as `/ucla/node1`. The links among network entities (routers and hosts) and their RIB tables are created using the NFD management tool `nfdc` provided with the NFD package. All NDN traffic are delivered directly on the Ethernet link layer with the MTU set to 9000. Link latency between routers are set according to the original downloaded topology data. We assume the link between a host and its router in a domain has zero latency.

All experiments have the number of blockchain nodes which is not greater than 200. The size is much smaller than the current Ethereum and Bitcoin networks which are usually in the orders of $1k$ nodes. Therefore, on both blockchain systems we set the maximum number of peers one blockchain node may have as 10 (the default value is 20), and the minimum number of peers for data pushing $maxpeerspush$ as 3 (the default value is 4). Increasing the second parameter help flood a new block more quickly at the cost of wasting more traffic. We reason that with a network of only 200 nodes and the parameter $maxpeerspush = 4$, it only takes 4 rounds of pushing for a block to reach $4^4 = 256$ nodes (with duplication) which covers the whole network with high certainty. It should not be the case in a large network where many nodes receive a block hash (then request the block) before receiving a pushed block (if any). Therefore, we set $maxpeerspush = 3$ to compensate for reducing of network size in our experiments.

5.2.2. Traffic utilization

We define traffic utilization as a ratio calculated by dividing the total traffic used by the network to produce a ledger by the storage size of saved information (the ledger). We differentiate two kinds of traffic: incoming traffic is the data that blockchain nodes receive from network; and outgoing traffic is the amount of data that blockchain nodes send to network. If nodes in a 100-node blockchain network send out a total amount of 500MB data and they can build a 2MB ledger, then the outgoing traffic utilization is $\frac{500MB}{2MB \times 100 nodes} = 2.5$. The ratio means that on average a node needs to send 2.5MB traffic to build $1MB$ of ledger data. A small number indicates efficient usage of traffic for ledger synchronization.

In this experiment, we create a blockchain network consisting of 100 nodes which are equally distributed on the 10 domains; thus, 10 blockchain nodes are running on a single physical server. Although `mininet` can handle a large network, we limit our experiment to the 100-size blockchain system due to the unstable performance of the NFDs. As the network size increases, we observe a significant number of Interest packets without response after four seconds. Handling timeout Interest packets creates extra traffic which could effect the accuracy of measuring of the traffic utilization of the blockchain system.

On each domain we create 10 hosts that all connect to the domain's router through a switch. On each host, we run a local NFD and blockchain clients. The NDN-based blockchain client connect to the NDN network through the local NFD on the same host. During a test, one of the blockchain node is

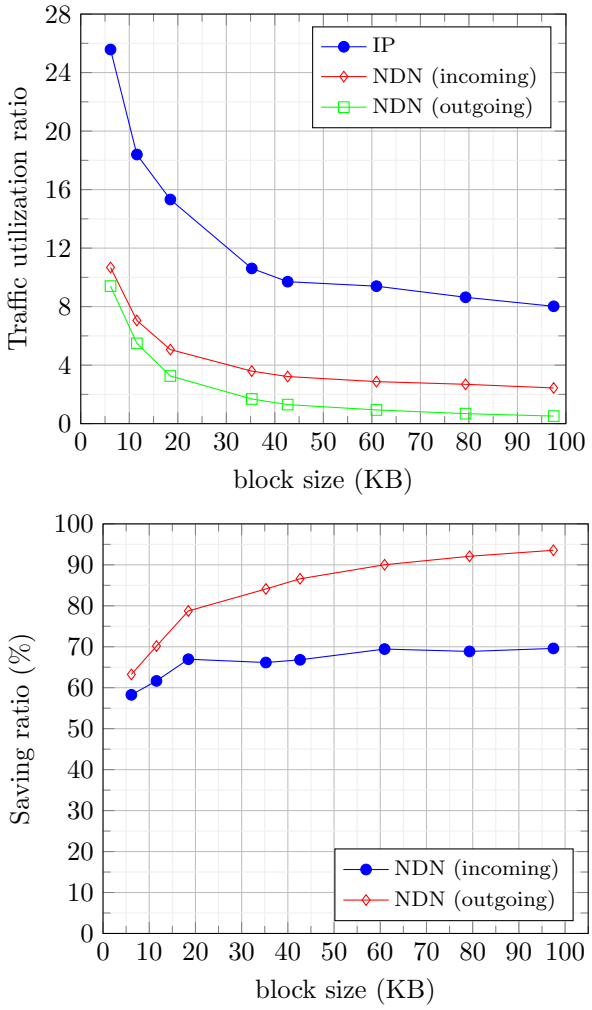


Figure 6: Comparing the traffic utilization of the two block chain systems

chosen as a bootstraping node from which all other remaining nodes attempt to query at the beginning of the Kademlia node discovery procedure. We let the systems stabilized for a certain period so that the nodes have enough peers before transaction and block propagation starts. Of all the blockchain nodes, we activate the mining feature on 10 nodes. The 10 miners are equally distributed (1 miner in 1 domain). We create another host on one of the domain for running a transaction generator. The generator sends transactions continuously to random blockchain nodes at a constant pace. We use linux network utilities to collect the traffic measures at the network interfaces of all hosts. The measure readings are the incoming traffic and outgoing traffic at the link layer; thus, they cover the blockchain data as well as the transport overhead (lower layers packet headers).

We run the above setting in 8 tests for each blockchain systems to investigate the traffic utilization at different transaction and block sizes. At each time, we inject a fixed-size bytes data into transactions to change their size. The size of injected data is varied from 0KB (no injection) to 2KB ([0KB, 0.1KB, 0.2KB, 0.5KB,0.7KB, 1KB, 1.5KB, 2KB]). Changing transaction size results in changing block size. At the end of each test, we estimate size of the ledger, average block size, and the traffic utilization ratios for the incoming traffic and outgoing traffic at the hosts.

In Figure 6 we plot the estimated traffic utilization ratios against the average block sizes (upper subplot). For IP-based blockchain, the total incoming traffic and the total outgoing traffic at all blockchain nodes should be the same as the network do not cache data. When block size value is small (due to small size of transactions), the ratios are high in both systems. The main cause is transport overhead as the application messages are small. As transactions size and block size get increased, the traffic utilization ratios on both system improve and get stabilized. But in all cases, the NDN-based blockchain outperforms the IP-based blockchain by a large margin. At the extreme case where the transaction size is 2KB and the average block size is 100KB, on average an IP-based blockhchain node need to send 8MB of traffic and receive the same amount to build 1MB ledger. Those numbers are reduced to only 2.5MB and 0.5MB, respectively, for an NDN-based blockchain node. It is also important to note that the outgoing traffic is much smaller than the incoming traffic on NDN-based blockchain due to the effect of network caching.

We next analyze how much traffic can be reduced by NDN-based blockchain. As the sizes of transactions and blocks increase, our protocol should utilize traffic more efficiently as the cost of gossiping announcement on P2P becomes less significant compared to the amount of announced data. We define a saving ratio as the percentage of traffic can be reduced when using NDN-based blockchain comparing to IP-based blockchain for producing the same-size ledger. The ratio can be estimated as:

$$Saving\ rario = \frac{IP\ traffic - NDN\ traffic}{IP\ traffic} \times 100\%$$

The lower subplot on Figure 6 shows the estimated saving ratios across different block sizes. Starting with transactions without injected data, the NDN-

based blockchain can reduce 58% and 63% the amount of sending and receiving data,respectively, compared to IP-based blockchain. As the transaction size and block size increase, the saving ratios increase as we expected. The values can reach up to 70% and 92% for incoming traffic and outgoing traffic, respectively.

5.2.3. Block propagation latency

The purpose of this experiment is to compare the block propagation time on the two blockchain systems. However, due to the limitation of the current implementation of NFD in terms of packet processing performance, measuring the block propagation latency needs to be conducted in settings where the workload on NFDs should be minimized. While preparing the experiments, we observe unstable performance of the current NFD implementation. In idling condition where there is no packet sending on the network, an NDN ping between two routers would cause extra about 1 millisecond delay compared to that of IP network. Under the condition where 10 miners is running on an NDN-based blockchain network of 200 nodes, we observe unstable latency measures with NDN ping messages while the latency on IP ping message remains stable.

Anticipating the undesirable performance of the NFD implementation, we made several adjustments from the previous experiment settings to guarantee the block propagation latency in NDN networks can be reliably measured. First, we reduce the number of NFDs running on a physical server to lessen the computational workload. It can be done by removing all local NFDs and letting the NDN blockchain clients on a domain connect to their router's NFD directly. As a result, only a single NFD on the router is running on a physical server. This change does not affect the latency measuring because in all experiments we assume zero latency for in-domain packet delivery. Second, we use only one miner to produce blocks in each running test since activating multiple miners would cause a large number of blocks created in a short period which in turn triggers the sending a large number of NDN packets for delivering these blocks to all the blockchain nodes. And lastly, we turn off the transaction propagation in the network. Instead, transactions are delivered directly to the single miner for building blocks. With these adjustment, the workload on NFDs can be significantly reduced which in turn allows measuring the block propagation latency in NDN-based blockchain network with fairness.

All experiments for measuring block propagation latency are conducted on a network of 200 blockchain nodes distributed equally on the 10 domains of the topology. Since the servers are on the same cluster in a local network, their time clocks are synchronized to have insignificant time differences. For each type of blockchain system, we run 10 experiments. In each experiment, we activate only one single miner to produce blocks. The miner locations are alternated through 10 domains in all experiments. During an experiment, a traffic generator residing at the same host of the miner sends $10,000$ transactions to the miner at a constant rate. At every blockchain clients we record the block size, the block delivery time and a flag indicating if it was the node that mines the block. At the end of the experiments, we collect the data from all blockchain nodes to estimate the propagation time of every blocks.

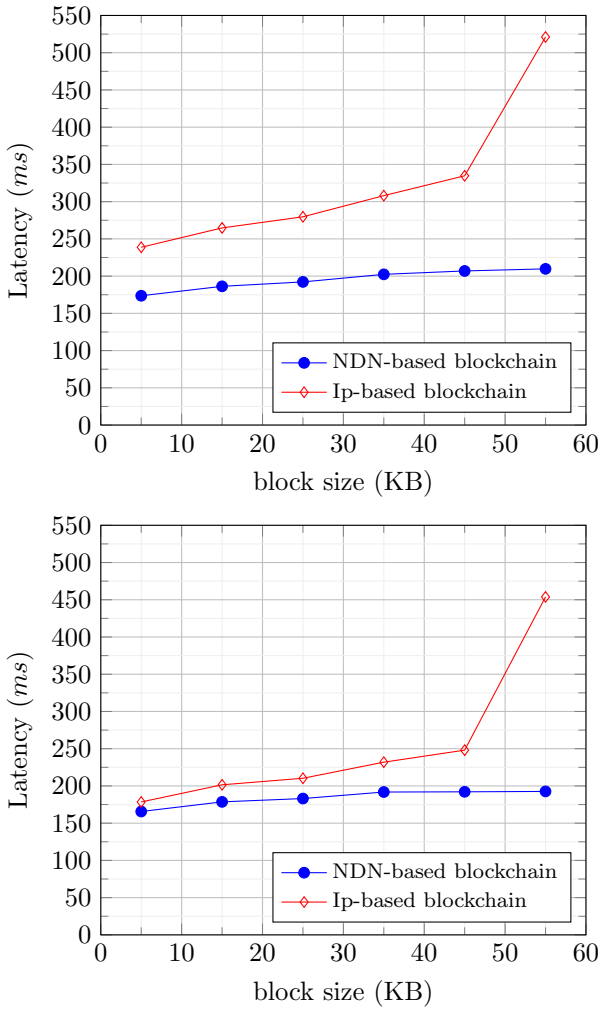


Figure 7: Mean (upper) / median (lower) block latency measures at different block size values

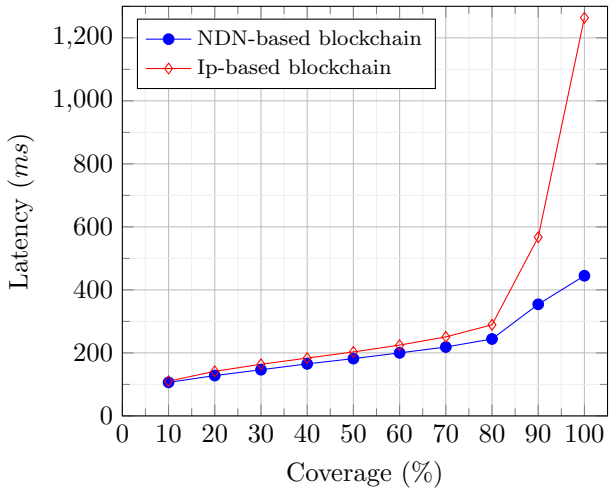


Figure 8: Block propagation latency at different network coverage percentages

Block propagation latency can be estimated from collected data by subtracting block created timestamps value at block miner from delivery timestamps at receiving node. Blocks which do not reach more than 90% number of nodes are discarded. We calculate average propagation latency for all the blocks. In addition, to show the influence of block size, we classify the blocks in six groups of different sizes up to 60KB. The outputs are plotted on on Figure 7.

The NDN-based blockchain outperforms the IP-based blockchain in all groups. We observe two interesting tendencies from the figure. First, the latency increases as the block size increase, however, not at the same pace in the two systems. The NDN-based blockchain performance is much less sensitive to the block size. The result explains the benefit of caching as a multi-segment block may need multiple rounds of message exchange; thus, having it at nearby NFDs may help reduce retrieving time significantly. Second, the block latency distribution in IP-based blockchain is skewed as there is a significant gap between the mean and median values. The result indicates that there are some nodes in the IP-based blockchain network receive block very late.

In the next analysis we take a closer look at how blocks are propagated on the network. We estimate the mean propagation latency for blocks to reach a certain percentage of network nodes. We plot the results against the network coverage percentage on Figure 8. Except for the first 10% nodes where the latencies are comparable in both systems, the NDN-blockchain outperforms in remaining parts. The difference gets larger as the coverage ratio increases. At the beginning, nodes in IP system would receive a new block more quickly through pushing mechanism. However, as time goes by, there are more nodes receiving the block by requesting (after receiving the block hash). At that point, the NDN-based blockchain system starts showing its advantage thanks to

caching mechanism of NDN. A dramatic increase of the latency occurs after the 80% coverage ratio in both networks. On average, blocks reach the last node after $444ms$ and $1260ms$ on NDN-based blockchain and IP-based blockchain respectively. The sudden jump is due to the fact that the experiment network topology has a isolated domain (the domain `/pku`) which has large link latencies to other domain. Hosts on that domain are receiving blocks late. The effect clearly shows on the IP-based blockchain as the latency increases significantly for those nodes. The effect is less dominant on the NDN-based blockchain as it needs only one blockchain node in the domain to have a block, the rest can retrieve the block from the NFD at the domain's router.

6. Related work

6.1. NDN-based data synchronization protocols

Since the inception of the NDN, there have been many works on the data synchronization problems due to their fundamental role in many practical applications such as group communication, file sharing. Chronosync [19], Vectorsync [21] and PSync [20] were introduced and they have become integral parts of the NDN client library for application development. The Chronosync implementation has been used in realizing NDN routing protocol [12].

All these protocols share some common properties: 1) participants have a common routable prefix for producing data; 2) synchronized data set is stored in a specialized structure so that the latest version (state) can be represented in a compact data object; 3) a new state is broadcasted to participants in an Interest packet; 4) Participants pull data items to synchronize whenever a new state is detected. For realizing the protocols, two conditions must be met: The common prefix is populated on the forwarding table of all NFDs; And the multicast forwarding strategy must be activated for the common prefix at all NFDs to support Interest broadcasting.

The works on data synchronization are different from our protocol for data propagation in blockchain systems in several aspects. First, in blockchain settings, the representation of system state has been defined by the Ethereum protocol suit and it has a specific role in blockchain application; thus, we can not apply the existing data synchronization protocols directly to our problem. Second, the existing protocols are only suitable for systems with known entities. That is, participating nodes in this protocol must have been known in advance. Usually, the number of participants is fixed or there must be a membership management module for monitoring the joining and leaving of the nodes. Third, the assumption made by these protocols is not practical for a public network as they requires a special policy treatment from the network operators: Participants must own a common routable prefix name for data producing; and a multicast forwarding strategy should be available at NFD routers for the given prefix. As a result, these protocols are not scalable to the extent of thousands of nodes as state-of-the-art Bitcoin or Ethereum networks. And lastly, these protocols were designed not for dealing with Byzantine failures which makes them not suitable for blockchain systems.

6.2. NDN-based blockchain

There are several attempts to implement blockchain systems on NDN. Block-NDN [9] is a bitcoin-like blockchain system that was developed to investigate the two questions: does blockchain technology fit better to NDN than IP? What are the advantages? It borrows the Chronosync protocol to broadcast blocks; therefore, it inherits all of its limitations. Although the experiment and analysis show that the system is better for data broadcasting in terms of message delay and traffic generation, the design is not suitable for practical deployments (See Section 6.1).

BoNDN [10] is another blockchain that is based on bitcoin implementation. The author identifies the main obstacle for the NDN-based blockchain design is switching from push-based model of IP to pull-based model of NDN to deliver data. Transactions are pushed to network using Interest broadcast. It is possible because the transaction size is small enough to be included within a single Interest packet. Blocks can have a significantly larger size, thus being delivered via a subscribe-push mechanism where miners subscribe to NFDs for publishing mined blocks.

The approach is simple and effective but it is not practical due to several issues. It remains doubtful if Interest broadcasting mechanism is possible in a public setting because it requires activating the multicast forwarding strategy at all NFDs. And, enabling the subscribe-push mechanism also needs modifications on both Interest packet format and NFD implementation.

Dledger [11] is an IOTA-based blockchain that was implemented for maintaining a solar electricity billing database. It is a permissioned blockchain whose participants must be managed. It uses Chronosync for data synchronization among blockchain nodes. Therefore, the system is not scalable and deployment would need a special policy from network operators. The Dledger was designed for a specific application. It is not a generic blockchain that can be used for different applications. The IOTA-based blockchain does not support smart-contract type transactions, thus having a very limited application. Our blockchain implementation is fully compatible with the state-of-the-art Ethereum blockchain. Any applications running on the current Ethereum network can be deployed on our blockchain system without modifications.

Comparing to existing approaches, the implementation of our protocol does not need any modifications to the existing infrastructure. Our approach tries to remain faithful to the decentralized-oriented design of blockchain systems while taking the most benefit from the good features of the NDN infrastructure.

6.3. Applications of blockchain technology to NDN

There have been many attempts to discover the applicability of blockchain technology for ICN in various topics. BlockAuth [28] framework was proposed to address the problem of insecure communication between producer and network forwarding plane in tracing-based producer mobility management. As the communication is unauthenticated, adversary can easily hijack prefix of the producer to mount various kind of attacks. The solution is to have a system to

manage producers and authenticate them at every routing update events. The BlockAuth is a distributed database in the form of a blockchain network composed of some capable core network routers. In [7] the author proposes Secure Blockchain-based Access Control (SBAC) framework which gives ICN content providers the control over sharing, auditing and revocation of their content. The blockchain technology is utilized as a decentralized database for recording the transactions. Other applications includes PKI system [6], securing NDN vehicular networks [29], cache poisoning defense and access control [30]. For a comprehensive review of the blockchain use cases in ICN, please refer to Asaf et al.[31].

The above applications were proposed to enhance some security aspects of ICN. However, the blockchain technology may have more important impacts on other areas in ICN. For example, it is foreseeable that the Internet Of Things (IOTs) and Vehicular Adhoc Network (VANET) will take huge benefits from the shifting paradigm of ICN thanks to its efficient content delivery and mobility supports. In these systems there is an astronomical number of connected devices that are deployed in heterogeneous and complex networks. This imposes many challenges on the devices management functionality. Immutable and trustless features of blockchain ledger can make it an attractive option for identity management and device authentication. Another use case can be a decentralized name resolution system, an alternative to the centralized NDNS, which map data content names to device locations to enhance network scalability and device mobility. A blockchain based data marketplace can be developed to enable the automatic data trading between devices. These are just a few standing out examples. With the introduction of our NDN-based Ethereum blockchain, we believe that many smart-contract enabled use cases [32],[33] which have been proposed so far can be adapted to the ICN networks as well.

7. Conclusion

This paper presents the design and implementation of an NDN-based Ethereum blockchain client. The development of the client serves two folds: providing a platform for fostering research and applications of blockchain technology on NDN; and enhancing the data delivery in blockchain with in-network caching and multicasting features of NDN. To improve the data delivery, the key idea was to use P2P overlay for announcing data availability while using NDN mechanism to retrieve data efficiently. Through experiments we show that the NDN-based blockchain system uses much less traffic compared to the conventional IP-based blockchain system. In addition, block propagation latency is also shortened on NDN-based blockchain thanks to data caching. Our blockchain client respects the specification of the Ethereum blockchain; thus, it is fully compatible with existing Ethereum-based decentralized applications. We plan to publish our implementation as open-source for further improvements and integration of new ideas by public research and developer communities.

8. Acknowledgement

This work was supported by the ICT R&D program of MSICT/IITP. [2017-0-00045, Hyper-connected Intelligent Infrastructure Technology Development]

9. References

References

[1] Bengt Ahlgren, Christian Dannewitz, Claudio Imbrenda, Dirk Kutscher, and Borje Ohlman. A survey of information-centric networking. *IEEE Communications Magazine*, 50(7):26–36, 2012.

[2] Athanasios V Vasilakos, Zhe Li, Gwendal Simon, and Wei You. Information centric network: Research challenges and opportunities. *Journal of network and computer applications*, 52:1–10, 2015.

[3] Lixia Zhang, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, KC Claffy, Patrick Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang. Named data networking. *ACM SIGCOMM Computer Communication Review*, 44(3):66–73, 2014.

[4] Dmitrij Lagutin, Kari Visala, and Sasu Tarkoma. Publish/subscribe for internet: Psirp perspective. In *Towards the Future Internet*, pages 75–84. IoS Press, 2010.

[5] Christian Dannewitz, Dirk Kutscher, Börje Ohlman, Stephen Farrell, Bengt Ahlgren, and Holger Karl. Network of information (netinf) – an information-centric networking architecture. *Computer Communications*, 36(7):721–735, 2013.

[6] Junjun Lou, Qichao Zhang, Zhuyun Qi, and Kai Lei. A blockchain-based key management scheme for named data networking. In *2018 1st IEEE International Conference on Hot Information-Centric Networking (HotICN)*, pages 141–146, 2018.

[7] Qiuyun Lyu, Yizhen Qi, Xiaochen Zhang, Huaping Liu, Qiuhua Wang, and Ning Zheng. Sbac: A secure blockchain-based access control framework for information-centric networking. *Journal of Network and Computer Applications*, 149:102444, 2020.

[8] Victor Ortega, Faiza Bouchmal, and Jose F. Monserrat. Trusted 5g vehicular networks: Blockchains and content-centric networking. *IEEE Vehicular Technology Magazine*, 13(2):121–127, 2018.

[9] Tong Jin, Xiang Zhang, Yirui Liu, and Kai Lei. Blockndn: A bitcoin blockchain decentralized system over named data networking. In *2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 75–80. IEEE, 2017.

[10] Jiang Guo, Miao Wang, Bo Chen, Shucheng Yu, Hanwen Zhang, and Yujun Zhang. Enabling blockchain applications over named data networking. In *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, pages 1–6, 2019.

[11] Zhiyi Zhang, Vishrant Vasavada, Randy King, and Lixia Zhang. Proof of authentication for private distributed ledger. In *Proceedings of the NDSS Workshop on Decentralised IoT Systems and Security (DISS)*, 2019.

[12] Lan Wang, Vince Lehman, A. K. M. Mahmudul Hoque, Beichuan Zhang, Yingdi Yu, and Lixia Zhang. A secure link state routing protocol for ndn. *IEEE Access*, 6:10470–10482, 2018.

[13] Alexander Afanasyev, Xiaoke Jiang, Yingdi Yu, Jiewen Tan, Yumin Xia, Allison Mankin, and Lixia Zhang. Ndns: A dns-like name service for ndn. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9. IEEE, 2017.

[14] Du Mingxiao, Ma Xiaofeng, Zhang Zhe, Wang Xiangwei, and Chen Qijun. A review on consensus algorithm of blockchain. In *2017 IEEE international conference on systems, man, and cybernetics (SMC)*, pages 2567–2572. IEEE, 2017.

[15] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[16] Petar Maymounkov and David Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.

[17] Christian Decker and Roger Wattenhofer. Information propagation in the bitcoin network. In *IEEE P2P 2013 Proceedings*, pages 1–10, 2013.

[18] Mauro Conti, Paolo Gasti, and Marco Teoli. A lightweight mechanism for detection of cache pollution attacks in named data networking. *Computer Networks*, 57(16):3178–3191, 2013. Information Centric Networking.

[19] Zhenkai Zhu and Alexander Afanasyev. Let's chronosync: Decentralized dataset state synchronization in named data networking. In *2013 21st IEEE International Conference on Network Protocols (ICNP)*, pages 1–10. IEEE, 2013.

[20] Minsheng Zhang, Vince Lehman, and Lan Wang. Scalable name-based data synchronization for named data networking. In *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, pages 1–9, 2017.

[21] Wentao Shang, Alexander Afanasyev, and Lixia Zhang. Vectorsync: distributed dataset synchronization over named data networking. In *Proceedings of the 4th ACM Conference on Information-Centric Networking*, pages 192–193, 2017.

[22] Spyridon Mastorakis, Alexander Afanasyev, Yingdi Yu, and Lixia Zhang. ntorrent: Peer-to-peer file sharing in named data networking. In *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–10. IEEE, 2017.

[23] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, pages 313–314, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[24] go-ethereum Authors. Go ethereum title. `https://github.com/ethereum/go-ethereum.git`, 2019.

[25] Junxiao Shi, Davide Pesavento, and Lotfi Benmohamed. Ndn forwarding at 100 gbps on commodity hardware. 7th ACM Conference on Information-Centric Networking (ICN 2020), Montreal, -1, 2020-09-30 2020.

[26] Faris Keti and Shavan Askar. Emulation of software defined networks using mininet in different simulation environments. In *2015 6th International Conference on Intelligent Systems, Modelling and Simulation*, pages 205–210, 2015.

[27] NDN project. Mini-ndn: A mininet based ndn emulator. `https://github.com/named-data/mini-ndn.git`, 2021.

[28] Mauro Conti, Muhammad Hassan, and Chhagan Lal. Blockauth: Blockchain based distributed producer authentication in icn. *Computer Networks*, 164:106888, 2019.

[29] Hakima Khelifi, Senlin Luo, Boubakr Nour, Hassine Moungla, Syed Hassan Ahmed, and Mohsen Guizani. A blockchain-based architecture for secure vehicular named data networks. *Computers & Electrical Engineering*, 86:106715, 2020.

[30] Kai Lei, Junjie Fang, Qichao Zhang, Junjun Lou, Maoyu Du, Jiyue Huang, Jianping Wang, and Kuai Xu. Blockchain-based cache poisoning security protection and privacy-aware access control in ndn vehicular edge computing networks. *Journal of Grid Computing*, 18(4):593–613, 2020.

[31] Khizra Asaf, Rana Asif Rehman, and Byung-Seo Kim. Blockchain technology in named data networks: A detailed survey. *Journal of Network and Computer Applications*, 171:102840, 2020.

[32] Shuai Wang, Liwei Ouyang, Yong Yuan, Xiaochun Ni, Xuan Han, and Fei-Yue Wang. Blockchain-enabled smart contracts: Architecture, applications, and future trends. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 49(11):2266–2277, 2019.

[33] Hong-Ning Dai, Zibin Zheng, and Yan Zhang. Blockchain for internet of things: A survey. *IEEE Internet of Things Journal*, 6(5):8076–8094, 2019.

10. Appendix

10.1. PoW consensus protocol

In a consensus protocol participating nodes cooperate to agree on an action. Once a decision is made, everyone follows. In the case of a blockchain system, the decision is the next block to be written. Node cooperation is enforced by either trust or incentive. The PoW consensus operates on the latter.

In PoW, nodes compete to solve serial computational puzzles for which winners are chosen as block creators and they receive rewards in the form of crytocurrency. Each puzzle is only solvable through brute-force efforts and its solution can be verified with insignificant computation.

Given a current ledger $\mathcal{L}_n$ of height n, and a pool of pending transactions $\mathcal{P}_n$, a node mines block $(n+1)^{th}$ as follows. The miner selects transactions from $\mathcal{P}_n$ to create an ordered list T_{n+1} based on his own criteria; however, the first transaction should credit himself with a reward. Next, he builds an unsealed block as $B^{\star}_{n+1} \leftarrow \texttt{BlockConstruct}(\mathcal{L}_n, T_{n+1})$, where `BlockConstruct` is a conceptual method for block creation conforming to a blockchain specification. The block is composed of a body b_{n+1} and a header $h^{\star}_{n+1}$. The method should put T_{n+1} into b_{n+1}, link b_{n+1} to $h^{\star}_{n+1}$, and link $h^{\star}_{n+1}$ to the last block of $\mathcal{L}_n$. The header $h^{\star}_{n+1}$ has an empty slot for PoW proof; thus, the block is so called 'unsealed'. The mining task is then to find an integer number $nonce_{n+1}$ that satisfies:

$$\texttt{Hash}(\texttt{BinEncode}(h^{\star}_{n+1},\, nonce_{n+1})) < \tau_{n+1}$$

where τ_{n+1} is an integer number called mining target for block $n+1$. The target is calculated with information from the ledger content. The only way to find a solution for the inequality is brute-force testing with different nonce values.

Once the proof $nonce_{n+1}$ is found, it is slotted into $h^{\star}_{n+1}$ to create a sealed header h_{n+1} which in turn replaces $B^{\star}_{n+1}$ with a sealed version B_{n+1} (composing of h_{n+1} and b_{n+1}). The node then extends its ledger as $\mathcal{L}_{n+1} \leftarrow \mathcal{L}_n, B_{n+1}$ and broadcasts the new block. Other nodes can check for correctness of the inequality with ease. If all transactions in B_{n+1} can be executed without failures, they accept B_{n+1} by extending their local ledgers with the block.

10.2. Kademlia DHT and peer discovery

A DHT system consists of connecting nodes that collectively maintain a *key-value* hash table. It supports two primitives: pushing a key-value pair to the table; and retrieving a value given its key. The system is designed to be scalable and resilient against churning. The principle to design such systems can be simply stated as: 1) putting node identities and data keys into a same key space; 2) using a key distance measure to partition the key space into partitions; 3) let a node manage keys in a partition which is close to its identity.

Implementing the two primitives requires an efficient algorithm to search for a node who manages a given key. The algorithm works in a collective manner.

When the key is not found in its local store, a node needs to look for help from the other nodes, and this procedure is executed iteratively. For carrying out this, the node must keep a routing table containing a list of other nodes that it may ask to query the key. The system is not scalable if the routing table has a complete list of nodes. Instead, the table only contains a few nodes to represent its view of the overlay network. The nodes are organized equally in buckets. A bucket represents a partition of the key space. The key space is partitioned in a way that the distances from the buckets to the identity of the routing-table's owner are sorted logarithmically increasing. By this method of network view representation, the collective search algorithm can operate in log(N) complexity where N is the total number of nodes in the system. For a full description of the Kademlia algorithm, please refer to Maymounkov et al. [16].

A joining node must known at least one existing peer on the system. It registers the peer as the first entry in its routing table then makes a query to the system using its own identity as input. As the search algorithm is executed collectively, the node learns of other existing peers and vice versa. When interacting with a previously unknown peer, a node may add the peer to its routing table. The updating operation may also happen whenever a user makes a query. By this way of working, the routing tables are gradually constructed.

Implementation of the Kademlia algorithm uses four RPC methods for communication between nodes: 1) `Ping`: learning peer's liveness. 2) `Store`: asking a node to store a *key-value* pair; 3) `Findnode`: node querying by its identity; 4) `Findvalue`: value querying by its key. The DHT algorithm does the dual jobs: maintaining the hash table (storing/retrieving key-value pairs) and constructing the P2P overlay structure (building routing table). Ethereum only utilizes it for the purpose of creating its P2P network. Therefore, the two methods `Store` and `Findvalue` operating on hash table are not implemented. The two remaining RPCs are implemented using UDP packets in Ethereum.

10.3. Peer management algorithm

The algorithm 4 presents methods that handle protocol messages and peer events at a node $p^\star$ for peer management. The node maintains a list trusted peer in a variable *peerlist*.

(Lines 6-15): Adding a new peer to the Kademlia routing table would generate an event triggering its handler at line 6. The node considers adding the peer owning rec_p to its trusted list. If rec_p is not in the blacklist and *peerlist* is not full, a handshaking procedure is conducted with the peer. The node creates a signed message $\langle \texttt{STATUS}, rec_{p^\star}, \texttt{true}, chainstatus^\star \rangle_{\delta_{p^\star}}$ for handshake offering and sends to the peer by calling `SendInterest`. Note that the node name embedded in rec_p is used for naming the sent Interest packet. The remote peer should responses with a Data packet d which encapsulates a handshake reply message $\langle \texttt{STATUS}, rec_p, accept, chainstatus_p \rangle_{\delta_p}$. The node considers its handshake is accepted if the following conditions are met: the reply message is verified; *accept* is `true`; and the returning $chainstatus_p$ matches to the chain

Algorithm 4 Peer management at node $p^\star$

Variables
  $peerlist$: list of trusted peers
  $chainstatus^\star$: chain status at $p^\star$
  $rec_{p^\star}$: node record of $p^\star$
  $prv_{p^\star}$: private key of $p^\star$
On receiving rec_p from Kademlia routing table **Do**
  check if $peerlist$ is not full
  $msg \leftarrow \langle \texttt{STATUS}, rec_{p^\star}, \texttt{true}, chainstatus^\star \rangle_{\delta_{p^\star}}$
  $d \leftarrow \texttt{SendInterest}(msg, rec_p.\texttt{Prefix}(), \texttt{false})$
  $reply = \langle \texttt{STATUS}, rec_p, accept, chainstatus_p \rangle_{\delta_p} \leftarrow$ decode d
  if $reply$ is verified
   && $accept$ is `true`
   && $chainstatus_p \equiv chainstatus^\star$ **then**
    create peer with rec_p and add to $peerlist$
  end if
On receiving a message $msg = \langle \texttt{STATUS}, rec_p, accept, chainstatus_p \rangle_{\delta_p}$
  verify msg, make sure $accept$ is `true`
      and $peerlist$ has no peer owning rec_p
  $decision$ = `false`
  if $peerlist$ is not full && $chainstatus_p \equiv chainstatus^\star$ **then**
    create peer with rec_p and add to $peerlist$
    $decision$=`true`
  end if
  $reply \leftarrow \langle \texttt{STATUS}, rec_{p^\star}, decision, chainstatus^\star \rangle_{\delta_{p^\star}}$
  $\texttt{SendInterest}(msg, rec_p.\texttt{Prefix}(), \texttt{false})$
On receiving a message $msg = \langle \texttt{BYE}, rec_p \rangle_{\delta_p}$ **Do**
  if msg is verified **then**
    drop from $peerlist$ peer owning rec_p
  end if
On application exiting
  for p in $peerlist$ **do**
    $msg \leftarrow \langle \texttt{BYE}, rec_{p^\star} \rangle_{\delta_{p^\star}}$
    $\texttt{SendInterest}(msg, p.\texttt{Prefix}(), \texttt{false})$
  end for

status of the node. If acceptance happens, the node creates a new peer with rec_p as input data, and then adds it to *peerlist*.

(Lines 16-25): In another event, when the node receives a handshake offering message $\langle \texttt{STATUS}, rec_p, accept, chainstatus_p \rangle_{\delta_p}$, it should process the message only if the following conditions are met: the message signature is verified; *accept* is `true` (for handshake offering); and no peer in *peerlist* owns rec_p (the peer is already accepted). If *peerlist* is not full and $chainstatus_p$ is matched to the chain status of the node, the handshake offer is accepted. Otherwise, it is rejected. If acceptance happens, the node creates a new peer with rec_p as input data, and then adds it to *peerlist*. Regardless of its decision, the node creates and signs a response message, and replies to the remote peer (line 24, 25).

(Lines 26-34): When the node is about to leave the system (line 30-34), it should notify its peers of its leaving by sending a $\langle \texttt{BYE}, rec_{p^\star} \rangle_{\delta_{p^\star}}$ message to each of them. On the other hand, if the node receives a message $\langle \texttt{BYE}, rec_p \rangle_{\delta_p}$ from a remote peer (line 26-29), it should drop from *peerlist* the owner of rec_p given that rec_p can be verified.

10.4. Data serving and fetching algorithm

The algorithm 5 presents simplified logic for serving a data segment and fetching a data item.

(Lines 1-8): This part implements a data serving handler which is triggered when a request for a data segment arrives. The request is a message $\langle \texttt{DATAREQ}, type, hash, segid \rangle$ which is extracted from an Interest packet. The node uses the type of data (*type*) and its hash value (*hash*) to search for the data item. If *type* is `TRANSACTION`, the node looks for a transaction from the transaction pool. Otherwise, it looks for a block from its ledger. It is also possible that the data can be retrieved from a cache table if it was requested before. If O is found, the node may cache it to serve subsequent requests. The number of segments and segment $segid^{th}$ of O are used to compose a response message $\langle \texttt{DATASEG}, wire, num \rangle$. The node then sends the message in a Data packet in response to the incoming Interest packet.

(Lines 6-29): This part implements `Fetch` method to fetch a data item. In general, the method is called whenever an data announcement is received. Its input arguments are extracted from the announcement. The arguments are: *type* is data type; *hash* is data hash value; *num* is number of segments; and $prefix$ is data owner's *nodename*.

Since it is possible that duplicated announcements are received, the method is called multiple times. Therefore, it always checks for the existence of a previous call, then waits for its outcome (lines 7-10). Otherwise, it moves on to the next steps where a worker method `DoFetching` is called once or twice to do the real fetching task.

In the first attempt, it tries to fetch the item using a public prefix for naming Interest packets (line 12) by setting the input *usehint* as `true`. Failing would indicate the data item is not available; thus, the owner of $prefix$ should be dropped from *peerlist* for not serving data. Otherwise the item is validated before it is returned to complete the method.

Algorithm 5 Data serving and fetching

On receiving $\langle$`DATAREQ`, $type, hash, segId\rangle$
  $O \leftarrow$ get item by $type$ and $hash$ ▷ may get from cache
  cache O ▷ if it was not cached
  $wire, num \leftarrow$ get segment $segId^{th}$ from O
  `SendData`($\langle$`DATASEQ`, $wire, num\rangle$)
procedure FETCH($type, hash, num, prefix$) ▷ fetch an item
  if item is being fetched **then**
    $O \leftarrow$ wait to receive from other thread
    return O
  end if
  $query \leftarrow (type, hash, num)$ ▷ build an item query
  $O \leftarrow$ `DoFetching`($query, prefix$, `false`) ▷ fetch with public prefix
  if O is `nil` **then**
    drop the owner of $prefix$ from $peerlist$
    return `nil`
  else if O is a valid item **then**
    return O
  end if
  $O \leftarrow$ `DoFetching`($query, prefix$, `true`) ▷ fetch with private prefix
  if O is `nil` **then**
    drop the owner of $prefix$ from $peerlist$
    return `nil`
  end if
  if O is not a valid item **then**
    drop the owner of $prefix$ from $peerlist$
    add $prefix$ to the black list
    return `nil`
  end if
  return O
end procedure
procedure DOFETCHING($query, prefix, usehint$) ▷ fetch segments
  $B \leftarrow []$ ▷ empty byte array of the item
  for $segId = 0$; $segId < query.num$; $segId$++ **do**
    $msg \leftarrow \langle$`DATAREQ`, $query.type, query.hash, segid\rangle$
    $d \leftarrow$ SendInterest($msg, prefix, usehint$)
    $\langle$`DATASEG`, $wire, num\rangle \leftarrow$ decode d
    $B \leftarrow [B\ wire]$ ▷ byte array concatination
  end for
  $O \leftarrow$ `BinDecode`(B) ▷ decode object from binary
  return O
end procedure

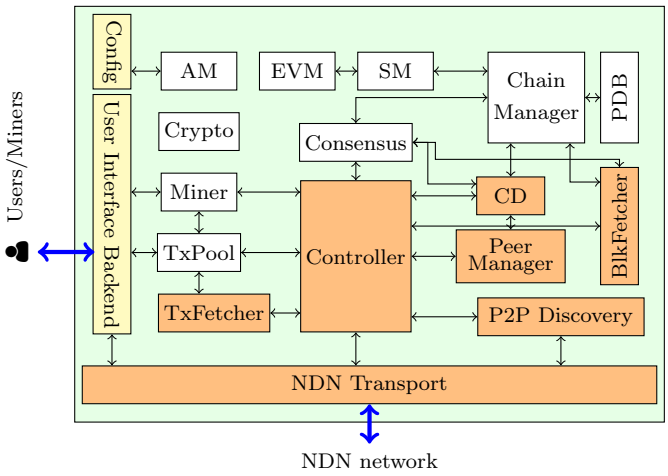


Figure 9: Functional architecture of the NDN-based Ethereum client

If the fetched item is invalid, it may be that a poisoning attack has occurred. In such a case, another fetching attempt is proceeded by calling the worker method `DoFetching` again. But this time the input *usehint* is set as `false` indicating that a private prefix should be used for packet naming. Failing to fetch the item would indicate that the owner of $prefix$ is not responsive, thus it should be dropped . In case where the received item O is invalid that peer should be added to the blacklist. Otherwise, O is returned to complete the procedure.

Lines (30-40): This part implements the worker method `DoFetching`$(query, prefix, usehint)$ to fetch a data item. The input *query* is used by a producer to search for the requested item. *query* is a tuple of values including a data type, a data hash, and a number of data segments. The input $prefix$ is *nodename* of data's owner. The input *usehint* indicates if $prefix$ is used as a forwarding hint (`true`) or as a part of packet name. The method fetches data segments iteratively in a loop. Inside the loop, a pair of `DATAREQ`/`DATASEG` messages are exchanged for retrieving a data segment. At the end, the item O is reconstructed and returned to the method's caller.

10.5. Functional architecture

In Figure 9, we show a complete list of functional components of the NDN-based Ethereum client. Components in white blocks are inherited without modifications. Components in orange blocks are implemented from scratch. And component in yellow blocks are inherited with modifications. In general, we completely remove the transport layer of `geth` and replace it with our implementation for NDN. Below we give a brief overview of each component.

Unmodified components:

- *Crypto* implements basic cryptographic primitives that are exclusively used by other components such as public/private key utility, hashing, signature signing/verifying.
- *Persistent Database (PDb)* provides interfaces for retrieving/storing the ledger and other data structures to persistent storage.
- *Account Manager (AM)* implements ethereum account management to enable wallet features such as account creation, deletion and monitoring.
- *Chain Manager (CM)* implements ledger data structures, provides interfaces for manipulating the ledger such as blocks/transaction querying and block insertion.
- *State Manager* implements the chain state data structure. It provides interfaces for querying historical state, updating new state by execution transactions, or rolling back to previous state.
- *EVM* implements the Ethereum Virtual Machine which executes transactions to update the chain state.
- *Transaction Pool (TxPool)* manages pending transactions that are buffered for including in the next mining block.
- *Miner* implements mining function. It collects new transactions from TxPool to build a new block then finding the POW solution for the block.
- *Consensus* implements the PoW consensus protocol. It also provides interfaces for creating/verifying/sealing blocks.

Modified components:

- *User Interface Backend (UIB)* implements backend services to support APIs for monitoring and controlling the blockchain client. It is modified to adapt to the replacement of P2P layer.
- *Config* implements parsing of client's command-line input arguments, configuration files, and blockchain genesis file. It is modified to replace IP network configurations with NDN configurations.

Replaced components:

- *NDN transport* maintains a connection to the local NFD and provides API interfaces to upper layer for NDN packet de/encapsulating, and NDN packet sending/receiving.
- *P2P* implements Kademlia DHT algorithm for peer discovery.
- *Peer Manager (PM)* implements the peer management function as explained in Section 10.3.

- *BlkFetcher & TxFetcher* implement block fetching and transaction fetching functions (see Sections 10.4, 4.5.6 and 4.5.7).
- *Chain Downloader (CD)* implements the blockchain downloading function. It periodically checks if the local ledger is behind ledgers of other peers. If so, it downloads missing blocks to update the ledger. This function is not described in this paper.
- *Controller* acts as an information hub for other modules. It handles all protocol messages sending/receiving and realizes the main logic of data propagation. It also provides interfaces for collecting/dispatching messages from/to other components including *BlkFetcher*, *TxFetcher*, *PM* and *CD*.