Idea: a general framework for decentralized applications based on the Bitcoin blockchain

Massimo Bartoletti, Davide Gessa, and Alessandro Sebastian Podda

Università degli Studi di Cagliari, Italy

Abstract. Bitcoin is a special-purpose decentralized application where mutually distrusting parties interact (without using a trusted third party) in order to trade digital currency. We build upon the Bitcoin protocol to realise a framework which programmers can extend to obtain *general-purpose* decentralized applications. We validate our framework by using it to implement (decentralized versions of) a key-value store, a message oriented middleware, and a middleware where the interaction among mutually distrusting parties is regulated by contracts.

1 Introduction

Decentralized applications (in short, *dapps*) are distributed applications implemented as peer-to-peer networks, wherein each node acts at the same time as client and as provider. A crucial feature of dapps is that they do not need a trusted authority to regulate the interaction among its nodes: suitable cryptographic protocols ensure trustworthy interactions among mutually distrusting nodes. To cope with failures, dapps usually record the historical data of all user messages in an immutable data structure stored by each node. So, if a node of the dapp disappears from the network, any other node can take its place without losing data, and without interrupting the service.

In the past few years, decentralized applications have gained wide popularity thanks to projects like *Bitcoin* [18]. Bitcoin is a dapp which allows mutually distrusting users to trade digital currency. All user transactions are stored in an immutable data structure called *blockchain*; the blockchain is updated with new blocks of transactions, using a distributed consensus process called *mining.* Mining enforces a chronological order in the blockchain, protects the neutrality of the network, and allows different users to agree on the state of the system. To be confirmed, transactions must be packed in a block that fits given cryptographic rules, verified by the network. These rules prevent previous blocks from being modified, since a change would invalidate all the subsequent blocks. Users can obtain a proof of existence for the messages sent to the blockchain (so to prevent e.g., their repudiation). Mining also creates a "competitive lottery", that prevents any user from easily appending new blocks to the blockchain. Hence, no user can control what is included in the blockchain, or replace parts of it to roll back their own spends.

The idea of using Bitcoin not just as a digital cash, but as the basis for applications beyond currency, is not completely new [8]. Indeed, the Bitcoin blockchain has been the foundation of several recent dapps [4,19,10,5,9]. E.g., *Namecoin* [4] uses the blockchain to create a domain registration mechanism, which is functionally equivalent to .com domains, but independent from ICANN and resistant to censorship. *CounterParty* [10] implements financial operations (like e.g., creation of virtual assets, payment of dividends, *etc.*) over the Bitcoin blockchain, by embedding its own messages in transactions. This has the drawback of forcing all Bitcoin nodes (even those not interested in financial operations) to download CounterParty-specific messages. *Blockstore* [5] is a decentralized key-value database. To allow only the interested nodes to download full data, Blockstore writes in the blockchain only the metadata (e.g., the hash of key-value pairs), while full data is stored in a distributed hash table (DHT).

While the dapps outlined before focus on specific problems, *Ethereum* [9] allows for developing *general-purpose* dapps. These dapps can be written in one of the Ethereum languages, compiled into bytecode instructions, and then interpreted by a decentralized virtual machine which runs over Ethereum nodes. Once a dapp is submitted to the Ethereum blockchain, it cannot be stopped or altered by anyone. Clients can use a dapp by sending a special transaction to an Ethereum node; this transaction includes the actual parameters for the invocation, and a certain amount of electronic cash to pay nodes for its execution.

Despite its elegance and generality, Ethereum has some drawbacks which may limit its practical applicability. Notably, programmers must write the whole dapp in one of the Ethereum languages, without exploiting existing legacy software or external (non-Ethereum) services. This constraint is required because, while computations carried on by a proper Ethereum virtual machine guarantee (with cryptographic proofs) to produce correct results, this cannot be ensured for other kinds of computation. Protecting clients from compromised Ethereum nodes, run by attackers who try to subvert legitimate computations, is another issue. In the absence of trust assumptions about the honesty of nodes, clients can only protect themselves against this kind of attacks by locally running their own node. However, running an Ethereum node is quite impractical, especially when the computing device has limited resources (e.g., power, bandwidth, disk space): indeed, one has to download the whole Ethereum blockchain, which today requires about 1.5GB of disk space, and takes about 2 hours to synchronize. Another limitation is that nodes cannot choose which dapps to run. Indeed, after a dapp is deployed on the Ethereum network, it cannot be modified anymore; the only way to update it is to broadcast a new dapp with the modified code, but the old version can still be used.

Contributions. We propose a framework to support developers in writing general-purpose decentralized applications which run over the Bitcoin blockchain. Our framework overcomes the drawbacks of Ethereum outlined above. In particular, we do not impose any constraint on the programming language used to write dapps: developers can invoke legacy applications or external services whenever they need to. Correctness of computations is guaranteed by a consensus mechanism: when a client queries a dapp, the query is replicated and sent to a set of distinct nodes; the actual result is obtained by taking the majority of the received answers. Similarly to Bitcoin and Ethereum, this mechanism is secure when the majority of the nodes is honest. The fact that correctness is obtained through the consensus mechanism, rather than through a decentralized virtual machine as in Ethereum, has an additional benefit: we only need to save in the blockchain the messages exchanged by clients, so avoiding the computational overhead of Ethereum, which must store in the blockchain the whole state of computations. This has also another benefit: each node in our framework can choose which dapp to execute. Indeed, to validate new blocks which appear in the blockchain, nodes no longer need to verify the state of *all* dapps. Consequently, clients do not need to run the blockchain daemon.

To validate the general applicability of our framework, we have used it to develop three real-world dapps. The first one implements a key-value database, with the same features as Blockstore [5]. The framework has been effective to reduce the programming effort: actually, while the original Blockstore implementation consists in $\sim$1600 LOC, ours only requires $\sim$100 LOC. The second dapp is a message-oriented middleware inspired to *RabbitMQ* [3]. Clients can use the middleware to exchange messages; to make the communication among clients asynchronous, the middleware stores unread messages in *FIFO* queues. As far as we know, ours is the first decentralized implementation of a message-oriented middleware over the Bitcoin blockchain (actually, RabbitMQ is distributed, but not decentralized). The last case study is far more complex, as it decentralizes the contract-oriented middleware in [6]. Clients advertise to the middleware *behavioural contracts* which describe their promised interaction protocol; the middleware dynamically composes clients with *compliant* contracts [7], by establishing sessions through which they can interact. The middleware monitors the interaction over sessions to detect contract breaches: if a client respects its contract, it will enjoy safe executions, otherwise it will be sanctioned.

We make available as open-source projects the framework (composed of a node daemon, a client library, and a Kademlia DHT) as well as all our case studies [13,14,15,12,16,11]. A tutorial on our key-value store dapp is in Appendix A.

2 System architecture and design

A decentralized application based on our framework is composed of a network of nodes. Assume that a developer wants to use the framework to build a dapp which offers some service through a set of APIs. To call these APIs, a client sends a message to a set of nodes which run the dapp. The framework then collects the responses from all the nodes serving the request, and it chooses the correct one through a consensus mechanism.

For those APIs which change the state of the dapp, the message is also published in a *public ledger*, accessible by all nodes of the network. In our framework, the public ledger is composed by the Bitcoin blockchain, and by a DHT [17]; in the latter we store full message data, to overcome the limit of 40 bytes for extra data in Bitcoin transactions. Messages stored in the blockchain are periodically

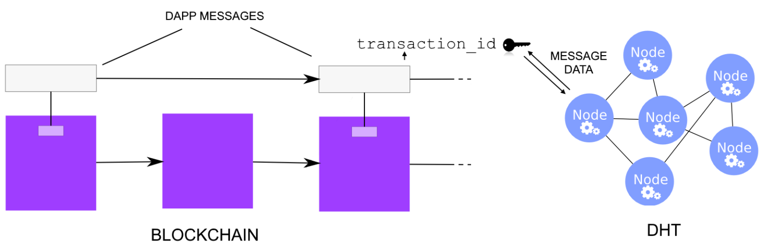

Fig. 1: Schema of data storage system, with the blockchain and the DHT.

read by nodes, to discover new requests for the dapp. The state of the dapp is stored internally by the nodes, and it is updated upon reception of such messages.

In what follows, we comment the main design principles of our framework.

Storage system. One of the key features of our framework is the storage system, which adheres to the principles of decentralized applications: it guarantees that messages cannot be deleted or modified by malicious users; further, users can obtain a *proof of existence* of the messages they send. To achieve these requirements, we build upon the Bitcoin blockchain [18], by using it as an *immutable* log of transactions.

Although the Bitcoin blockchain is primarily intended to trade digital cash, the protocol allows to include some extra data. To reduce spam transactions in the blockchain, the size of extra data in a transaction cannot exceed 40 bytes[1]. To overcome this limit, we use the blockchain for saving message metadata only (hereafter referred to as *headers*), while we exploit a DHT to store the full message data. The scheme of the data storage system is illustrated in Figure 1. In our framework, we implement *Kademlia*, a popular DHT also used by *eMule* [2] and *bittorrent* [1].

Headers include various metadata: a magic flag for message recognition, the message hash, the dapp identifier, and the unique identifier of the Bitcoin transaction (used to retrieve the full message data from the DHT). We also include the hash of the data stored in the DHT, to protect against data tampering attacks. Finally, the headers identify the author of the message.

Client application. The framework client library allows clients to invoke nodes through standard web-based calls (e.g., JSON-RPC). If a client invocation requires to change the dapp state, the client must also publish a suitable message in the public ledger, following the protocol sketched in Figure 2b. First, the client sends a raw message m to the node, which generates an unsigned transaction

[1] In the most recent version of the protocol this limit is 80 bytes; we stick to the old limit of 40 bytes for backward compatibility.

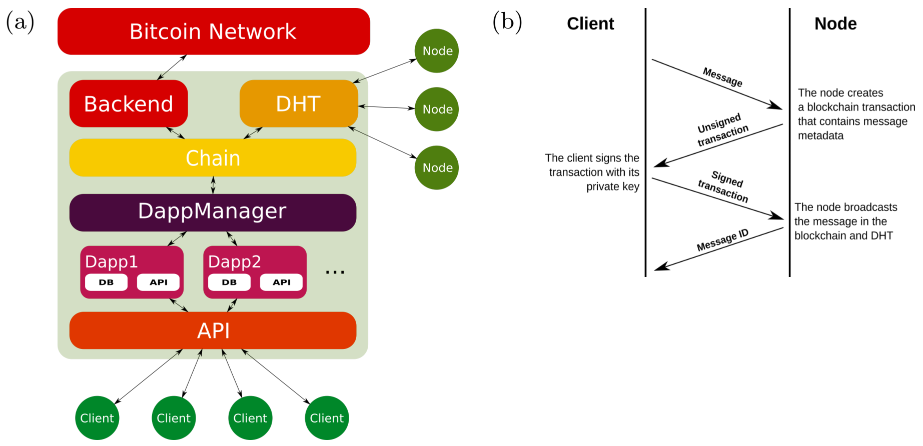
(a)
Bitcoin Network
Backend
DHT
Chain
DappManager
Dapp1
DB
API
Dapp2
DB
API
...
API
Node
Node
Node
Client
Client
Client
Client
(b)
Client
Node
Message
The node creates a blockchain transaction that contains message metadata
Unsigned transaction
The client signs the transaction with its private key
Signed transaction
The node broadcasts the message in the blockchain and DHT
Message ID

Fig. 2: (a) components of the node daemon; (b) message publication protocol.

t_m^{uns} for that message. Then, the node returns t_m^{uns} to the client, who signs it using its Bitcoin private key. Once the node receives the signed transaction t_m^{sig}, it broadcasts the header to the blockchain, and the full message data to the DHT. The publication protocol prevents malicious nodes from performing tampering and replay attacks. For tampering attacks, note that modifying client messages would invalidate the signature and the hash; for replay attacks, note that the Bitcoin protocol does not allow to publish two transactions with the same hash.

Node daemon. The node daemon follows the schema illustrated in Figure 2a.

The `Backend` module implements the Bitcoin protocol and it handles new blocks and transactions; when a new block is discovered, the `Chain` module retrieves all the associated transactions, and scans them to extract those messages which contain the framework headers. If the header of message m is found in a transaction with identifier k, the `Chain` module retrieves the content of m from the DHT, using k as key. The content is then checked for validity (hash, signer and size match the data in the DHT), and passed to the `DappManager`, which looks for a suitable dapp to handle the message.

3 Validation

To validate the general applicability of our idea, we have developed three case studies. They show that our framework is suitable for developing complex and reliable decentralized applications, and, at the same time, it helps to reduce the programming effort and the issues related to centralization.

Key-value store. The first case study implements a key-value storage service, similar to Blockstore [5]. This dapp offers two APIs: `set(key,value)` and `get(key)`. A `set` request saves a new immutable key-value pair in the storage, while `get` allows to retrieve a value previously set. Since `set` changes the state of the dapp, this API requires the client to publish a message in the blockchain. Nodes scan the blockchain for new messages and, when they find the `set` request, they save the new key-value pair in their local database. Executing `get` does not require to update the blockchain, because nodes are able to handle this request internally.

This case study highlights two main advantages of our framework. First, the use of the blockchain guarantees that a key already set by an user cannot be replaced or deleted by another one. Second, the programming effort is reduced (Blockstore has ∼1600 lines of source code, while our dapp takes ∼100 lines). Appendix A shows a detailed walk-through of the code of this case study.

Message-oriented middleware. The second case study is a message-oriented middleware (MOM), i.e. a message broker through which clients can communicate by sending/receiving messages to/from *FIFO* queues. Our dapp implements a subset of the features of *RabbitMQ* [3], a widespread distributed MOM. RabbitMQ allows instances of the MOM to join, by forming a federation of brokers which act as a single logical broker. This helps in dealing with failures, since the state of the queues is replicated. Actually, RabbitMQ federated brokers suffer of trust centralization, because they are still controlled by a single party. Were this not the case, a malicious federated broker could act as a Dolev-Yao attacker, by modifying, dropping or rerouting client messages. Note that RabbitMQ cannot give any guarantee about exchanged messages, since their existence and consistency are not (cryptographically) proved. By contrast, our dapp ensures that all messages are correctly dispatched, without assuming any broker to be trusted.

Contract-oriented middleware. The last case study is a contract-oriented middleware, i.e. a MOM where the interaction among users is regulated through *contracts* which formally specify their interaction behaviour [6]. Users advertise contracts when they want to establish sessions with other (unknown and untrusted) users; the middleware creates session among users with *compliant* contracts [7]. The interaction in each session consists in sending and receiving messages, and it is monitored by the middleware to detect contract violations (e.g., when some action is not performed when prescribed by the contract).

Figure 3 illustrates the main features of this middleware. In (1), users A and B advertise their contracts; in (2), the middleware determines that these contracts are compliant, so it establishes a session through which A and B can interact. In (3), A sends to the middleware a message for B, which then collects it. In (4) the monitor detects an attempt of B to do some illegal action.

The correctness of a contract-oriented middleware relies on the assumption that the state of sessions is recorded correctly. This is a crucial task, since sessions are inspected to assign blame to users who violate contracts, and coherently sanctions them. Centralized implementations of the middleware (like the one in [6]), have a main drawback: since users must trust a third party to record the state of sessions, an attacker which corrupts that party may break the correctness of the whole system. Distributed (but not decentralized) implementations have a

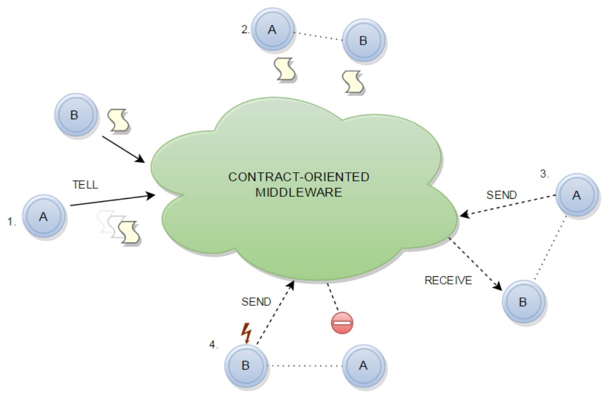

Fig. 3: Basic interactions in a contract-oriented middleware.

different drawback: to securely distribute the state of sessions among nodes in the network, suitable cryptographic protocols have to be devised. Our decentralized implementation solves both problems: every node of the network records the state of all sessions (which is correct because in the public ledger); further, the influence of malicious nodes is mitigated by exploiting the consensus mechanism.

4 Conclusions

Our idea is to leverage on the Bitcoin blockchain in order to realize general-purpose decentralized applications. To foster this idea, we have implemented a framework which supports developers in this task: notably, no familiarity with the Bitcoin protocol or peer-to-peer networks is required to developers, who can program in the same way as they would do for centralized applications. Further, decentralization is transparent to clients: when invoking a dapp in our framework, one cannot distinguish whether its request is served by a peer-to-peer network, or by a single node. Applications built upon our framework avoid the classic issues of the centralized model: for instance, in a centralized architecture clients must rely on a trusted third party; if this gets damaged or attacked, the system becomes unreliable. Conversely, the dapps running over the nodes of our framework are reliable whenever the majority of these nodes are honest.

Our proposal addresses the main issues of Ethereum [9] which, at the time of writing, is the only other tool supporting general-purpose dapps. Specifically, in our framework: dapp developers can choose their preferred programming language, use legacy software or external services (at the cost of enlarging the trusted computing base); the Bitcoin blockchain is not overloaded, since we only use it to store metadata; also clients with limited computational resources can use dapps run over our framework; obsolete/flawed dapps can be updated or even removed (note however that this is considered a *mis*feature in the Ethereum community). The crucial design choice made to obtain these features was using a consensus mechanism to ensure consistency of computations, instead of decentralizing the state of computations as in Ethereum. Overall, we think that these features make our framework more suitable than Ethereum for most practical uses. The validation step reported in Section 3 has highlighted that the abstraction layer offered by the framework helps programmers in simplifying the development of dapps (e.g., in terms of source lines of code).

In our framework, clients pay a small fee ($\sim$0.01EU) to call APIs which change the state of a dapp (this is due because any message which changes the state of a dapp is recorded in a transaction in the Bitcoin blockchain). However, these fees do not serve as an incentive for nodes which run dapps, since they can be collected by miners outside the framework. This is a bit unfair, since nodes must consume computational resources to execute dapps. We think that some form of incentive for nodes would make the framework more effective: for instance, we could impose an additional fee for clients, to be split among nodes which serve their requests (according on their computational load). Another possible extension would be to create a lottery mechanism — similar that of Bitcoin mining — where nodes receive a reward with a certain probability, proportional to the number of dapps they run.

References

1. bittorrent. `http://www.bittorrent.com`.
2. emule. `http://www.emule-project.net`.
3. Rabbitmq. `https://www.rabbitmq.com`.
4. Namecoin: a decentralized DNS service. `https://wiki.namecoin.org`, 2011.
5. Blockstore: Key-value store for name registration and data storage on the Bitcoin blockchain. `https://github.com/blockstack/blockstore`, 2014.
6. M. Bartoletti, T. Cimoli, M. Murgia, A. S. Podda, and L. Pompianu. A contract-oriented middleware. In *Proc. FACS*, 2015.
7. M. Bartoletti, T. Cimoli, and R. Zunino. Compliance in behavioural contracts: a brief survey. In *PLABS*, volume 9465 of *LNCS*. Springer, 2015.
8. J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten. SoK: Research perspectives and challenges for bitcoin and cryptocurrencies. In *IEEE Symposium on Security and Privacy*, pages 104–121, 2015.
9. V. Buterin. Ethereum: a next generation smart contract and decentralized application platform. `https://github.com/ethereum/wiki/wiki/White-Paper`, 2013.
10. R. Dermody, A. Krellenstein, O. Slama, and E. Wagner. CounterParty: Protocol specification. `http://counterparty.io/docs/protocol_specification/`, 2014.
11. D. Gessa. Blockstore case study. `https://github.com/contractvm/cvm-dapp-blockstore`.
12. D. Gessa. Contract-oriented middleware case study. `https://github.com/contractvm/cvm-dapp-cotst`.
13. D. Gessa. Contractvm daemon. `https://github.com/contractvm/contractvmd`.
14. D. Gessa. Contractvm library. `https://github.com/contractvm/libcontractvm`.
15. D. Gessa. Kad.py Kademlia DHT. `https://github.com/dakk/kad.py`.
16. D. Gessa. Message-oriented middleware case study. `https://github.com/contractvm/cvm-dapp-fifomom`.

17. P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Peer-to-Peer Systems*, pages 53–65, 2002.
18. N. Satoshi. Bitcoin: a peer-to-peer electronic cash system. `https://bitcoin.org/bitcoin.pdf`, 2008.
19. S. Wilkinson, J. Lowry, and T. Boshevski. Metadisk: a blockchain-based decentralized file storage application. `http://metadisk.org/metadisk.pdf`, 2014.

A Dapp source code walk-through

This appendix is a brief tutorial for dapp developers. We illustrate how to write a dapp by commenting the source code of our first case study, i.e. the decentralized key-value storage service. The full source code of this dapp is available at [11].

The key-value storage dapp offers two APIs: `set(key,value)` and `get(key)`. A `set` request saves a new immutable key-value pair, while `get` retrieves a previously set value. Since `set` changes the state of the dapp, this API requires the client to publish a suitable message in the blockchain. Nodes scan the blockchain for new messages, and when they find a `set` request, they save the new key-value pair in their local database. Instead, executing a `get` does not require to publish any message, because nodes can handle this request internally.

The dapp is split in two main parts: the first (Listings 1 to 4) run in nodes, while the second (Listing 5) implements the library that client applications use to interact with the dapp. Note that, while we have chosen Python for developing this dapp, our framework supports arbitrary programming languages. The mechanism used to this purpose is standard: the programmer codes the core features of dapp in her preferred language, and the framework uses them through the foreign function interface.

A.1 Node part

In Listing 1 we define the protocol and the supported messages of the dapp. At lines 1-4 we define a set of constants containing the code for each type of message and the dapp code. Then we extend the `Message` class, by defining a constructor for the `set` message (lines 7-13), and by overriding the function `toJSON()` for the serialization of message data.

The next step is to write the core of our dapp: this is done in Listing 2 by extending the class `dapp.Core`. In this class we define all the methods that interact with the dapp state, including query and pair insertion. We define a function to obtain a value given its key, and another one to set a new key-value pair. We save key-value pairs in the internal database which is automatically created by the framework to store the state of a dapp.

The services offered by the dapp are exposed to client applications as APIs. These APIs are implemented in Listing 3, where we extend the class `dapp.API`, and we create a `dict` object which contains new API calls (lines 5-6). Then, we write our two APIs:

- `set (key,value)`: creates a `set` message with a new key-value pair, and returns message broadcasting information;
- `get (key)`: gets a value for a given key, by invoking the `Core.get` method.

Finally, in Listing 4 we bind all the classes created so far. We use the method `handleMessage` (lines 10-12) to tell the `DappManager` how to handle messages.

```
class BlockStoreProto:
  DAPP_CODE = [ 0x01, 0x02 ]
  METHOD_SET = 0x01
  METHOD_LIST = [METHOD_SET]

class BlockStoreMessage (Message):
  def set (key, value):
    m = BlockStoreMessage ()
    m.Key = key
    m.Value = value
    m.PluginCode = BlockStoreProto.DAPP_CODE
    m.Method = BlockStoreProto.METHOD_SET
    return m

  def toJSON (self):
    data = super (BlockStoreMessage, self).toJSON ()
    if self.Method == BlockStoreProto.METHOD_SET:
      data['key'] = self.Key
      data['value'] = self.Value
    else:
      return None
    return data
```

Listing 1: Blockstore dapp: messages.

A.2 Library

In Listing 5 we define a library module, that binds the API calls described in Listing 3 inside a library, which will be used to write client applications. We do this by extending the `DappManager`. This class includes the services of our dapp, by binding the API calls `bs.get` and `bs.set`. The method `set` only creates and broadcasts a new message containing the given key-value pair; the method `get` performs a consensus query to nodes, and returns the resulting value.

A.3 Example usage

Listing 6 shows a simple “hello world” client of our Blockstore dapp. At lines 4-5 we create a `ConsensusManager`, and we bootstrap it with a seed node. At lines 7 we create a `Wallet` object, by using an external service with private keys saved in the file `app.wallet`. At line 8 we create a `BlockstoreManager`, by using the `ConsensusManager` and `Wallet` objects created before. At lines 10-11 the script asks the user for a key-value pair, and at line 13 it publishes it to the framework. Then, at line 14-15 the script asks the user for a key, and then queries and returns the associated value (if any).

```python
class BlockStoreCore (dapp.Core):
  def __init__ (self, chain, database):
    super (BlockStoreCore, self).__init__ (chain, database)

  def set (self, key, value):
    if self.database.exists (key):
      return
    else:
      self.database.set (key, value)

  def get (self, key):
    if not self.database.exists (key):
      return None
    else:
      return self.database.get (key)
```

Listing 2: Blockstore dapp: Core.

```python
class BlockStoreAPI (dapp.API):
  def __init__ (self, core, dht, api):
    self.api = api
    rpcmethods = {}
    rpcmethods["get"] = { "call": self.method_get }
    rpcmethods["set"] = { "call": self.method_set }
    errs = { "KEY_ALREADY_SET": {"code": -2, "message": "Already set"},
             "KEY_IS_NOT_SET": {"code": -3, "message": "Is not set"} }
    super (BlockStoreAPI, self).__init__(core, dht, rpcmethods, errs)

  def method_get (self, key):
    v = self.core.get (key)
    if v == None:
      return self.createErrorResponse ("KEY_IS_NOT_SET")
    else:
      return v

  def method_set (self, key, value):
    if self.core.get (key) != None:
      return self.createErrorResponse ("KEY_ALREADY_SET")

    message = BlockStoreMessage.set (key, value)
    return self.createTransactionResponse (message)
```

Listing 3: Blockstore dapp: API.

```python
class blockstore (dapp.Dapp):
  def __init__ (self, chain, db, dht, apiMaster):
    self.core = BlockStoreCore (chain, db)
    api = BlockStoreAPI (self.core, dht, apiMaster)
    super(blockstore, self).__init__ (      BlockStoreProto.DAPP_CODE,
                                            BlockStoreProto.METHOD_LIST,
                                            chain, db, dht, api)

  def handleMessage (self, m):
    if m.Method == BlockStoreProto.METHOD_SET:
      self.core.set (m.Data['key'], m.Data['value'])
```

Listing 4: Blockstore dapp.

```python
from libcontractvm import Wallet, ConsensusManager, PluginManager

class BlockStoreManager (DappManager.DappManager):
  def __init__ (self, consensusManager, wallet = None):
    super (BlockStoreManager, self).__init__ (consensusManager, wallet)

  def set (self, key, value):
    cid = self.produceTransaction ('bs.set', [key, value])
    return cid

  def get (self, key):
    req = self.consensusManager.jsonConsensusCall ('bs.get', [key])
    return req['result']
```

Listing 5: Blockstore dapp: library.

```python
from libcontractvm import *
from blockstore import BlockstoreManager

consMan = ConsensusManager.ConsensusManager ()
consMan.bootstrap ("http://192.168.1.102:9095")

w = WalletExplorer.WalletExplorer (wallet_file="app.wallet")
bs = BlockStoreManager.BlockStoreManager (consMan, wallet=w)

ykey = input ('Insert a key to set: ')
yvalue = input ('Insert a value to set: ')
bs.set (ykey, yvalue)

ykey = input ('Insert a key to get: ')
value = bs.get (ykey)
print (ykey,'=',value)
```

Listing 6: Example usage of the client library for the Blockstore dapp.