Bitsquare P2P network

Version 0.8

- Overview
- Connections
- Startup
  - Tor node created
  - Request data map
  - Hidden service created
  - Authentication
    - Authentication handshake protocol:
- Peers
  - Connection maintenance
- Data storage
  - Storage data protection
  - Private data protection
  - Time to live
  - Data re-publishing
  - Data types used for storage
  - Data size
- Flooding algorithm
- Direct connections
- Seed nodes
- Attacks
  - Spam/DDoS
    - Protection
- Simulations
- Bitcoins P2P network
    - References regarding Bitcoin P2P network
- Open questions

Please note:
That document is not updated to the current implementation. There have been quite a few changes recently (removed the authentication handshake)

Overview

Bitsquare uses a native Tor binary as proxy to the network. This is integrated into the application so the user doesn't need to do any setup or manually startup Tor.
Each node starts a Tor hidden service and uses its onion address as its P2P network address.
There are several seed nodes which are also operated as hidden services used for bootstrapping into the network.
Data which needs to be stored in the network is broadcasted to all connected peers using a flooding (gossiping) algorithm similar to that used in the Bitcoin P2P network.

Data can be public (offers) or private (mailbox messages). All private data is encrypted and signed. Ownership protection of data is achieved with signatures.
Every node checks if the hash of a part (prefix) of the onion address of the receiver in the data matches his address and if so he tries to decrypt the private data (similar to BitMessage). The blur factor of the address hash is a compromise between privacy protection and performance optimization (avoiding to need to try to decrypt all data).
Trade messages are sent directly (encrypted and signed) to the other peer.
All data has a time to live and every node checks periodically if stored data is expired and if so removes that data from its local storage.

As the publishing of the hidden service takes about 40 sec. we can use that as a kind of protection against Sybil attacks (similar to PoW). We use an authentication scheme to give only full access to the network after a node is authenticated (reachable with his onion address) by other nodes. Storage operations and direct messages are only accepted after the connection is authenticated. Only an initial GetData call is allowed without authentication to be able to display already data we want to display when starting up Bitsquare (e.g. offers in the offer book).
Additionally there are limits for the size and frequency of messages as well as checks for the expected data type, and a P2P network ID derived from the Bitcoin network type to avoid overlaps between testnet, mainnet and regtest.

Please note:
The P2P network is still in development and recent changes are not reflected here. There are also not all edge cases covered here to limit the complexity.
Please check out the Source code if you want to get the full picture.

Connections

Each node runs a socket server for creating new inbound connections and maintains a collection of outbound connections. For the application it is transparent if the connection to a peer is an inbound or an outbound connection. Messages can be sent on either type of connection by writing into the output stream. Same for receiving messages, they can be delivered by either an inbound or an outbound connection. Only one connection is maintained to each other peer.
Graceful disconnection (at shutdown) is done by sending a CloseConnectionMessage and waiting a short period before closing the connection.

How to deal with clients which are going into standby/hibernate mode is an open question to be investigated. Network connections will be halted but the client does not detect that yet so we cannot remove an offer (which requires that the publishing node is online) as it is the case when shutting down the app.
We should find out how to get the OS event before hibernate starts to allow the application to remove the offers, as well when hibernate stops to re-publish the offers.
Maintenance tasks help to detect sleeping nodes. Time to live ensures inactive offers will not stay too long as zombies in the offer book. A zombie offer only causes inconvenience as the taker will detect quickly that the offerer is offline.

UPDATE:
The current version is handling sleep/hibernate/lost connection situations.

- The connection socket timeout is reduced to 1 min. To find faster when a peer is not sending/responding to keep alive messages.
- There is an “check for inactivity” timer (Clock class, used in PeerManager) which detects if the app thread was idle for more than 5 seconds and sends a wakeup event when that idle time has ended (when sleep mode is over).
- There is still no OS system notification implemented, but should be possible and might get implemented if needed later. The codebase of IntelliJ community edition (OS) has solutions for such OS events as far I saw.

Startup

Tor node created

To create a tor node takes about 6 seconds.
Here the native Tor binary is started and the proxy (Sock5Proxy) is setup.
For the Tor proxy implementation we use the JTorProxy, JTorCtrl and JSocks libraries.

Request data map

As soon the node is connected to the Tor network the user can request data from a randomly chosen seed node to get the initial data set (we want to display offers in the offerbook). If the chosen seed node does not respond we try the next one. If none succeeds we repeat after a random pause and try to use previously persisted known peers for the initial data request.

As soon the data is available we leave the splash screen in the UI and we are ready for starting authentication to the connected seed node as soon as the hidden service is ready.

Hidden service created

After the creation of the hidden service it will be published to the Tor directory servers which takes about 30-40 sec. until it is available. After that the node contacts the connected seed node to perform the initial GetData request.

Authentication

We start to authenticate first to the already connected seed node.
Authentication gives a proof that the self reported onion address is correct and can be reached by the other peer. The creation of a hidden service takes 40 sec. and can be used as a kind of Sybil protection to avoid network-level attacks. Authentication makes sure that this PoW-like Sybil protection is not circumvented.

We create an outbound connection (or use the existing in case of the first seed node) and send an AuthenticationRequest message. The receiving peer disconnects then that connection and starts a new connection with our reported onion address. We use nonces for verification that the senders and receivers are as expected.

In the authentication handshake we exchange our peerAddresses (reported peers from other nodes and connected nodes) with the other node.

Authentication handshake protocol:

1. Node 1 creates a requesterNonce and sends AuthenticationRequest with requesterNonce and his own address to node 2.
2. Node 2 receives AuthenticationRequest and shut down connection. After shutdown is completed he creates a responderNonce and sends an AuthenticationResponse to node 1 with both requesterNonce and responderNonce and his own address.
3. Node 1 receives AuthenticationResponse and verifies both requesterNonce and address of node 2 (reported address need to be same as initially used one). Next he sends a GetPeersAuthRequest with his address, the responderNonce and the peerAddresses to node 2. The peerAddresses is the collection of reported peerAddresses and connected peers, excluding its own address. If the message sending succeeded he stores the peers address in the connection.
4. Node 2 receives GetPeersAuthRequest and verifies both nonce responderNonce and address of node 1. If valid he sends a GetPeersAuthResponse with his address and peerAddresses set to node 1. After that he adds the peerAddresses from the GetPeersAuthRequest to his reported peerAddresses. When message sending succeeded he adds the peer to his connected peers and mark the connection as authenticated. The authentication handshake is completed for his side.
5. Node 2 receives GetPeersAuthResponse and adds the peerAddresses to his reported peerAddresses, adds the peer to his connected peers and mark the connection as authenticated. The authentication handshake is now also completed for his side.

The handshake is now completed. In case it was the first authentication we request again the data set as we could have missed data updates in the meantime.
We continue with the next authentication handshake as long as the number of connections is lower than the target (10 connections), otherwise we shut down the connection with the least recent activity timestamp. The activity timestamp is updated to the current time at every data send or receive on the connection.

The next peers will be from the reported addresses set. Only in case we don't have enough reported peers we go back to the seed nodes.
If the authentication fails and we don't get our target of 10 connections we repeat after a random delay (1-2 min.).

If the local neighbor set is larger than 1000 nodes we remove randomly the exceeding nodes from the reported neighbor set.

Peers

We maintain a set of 10 peer connections. We choose first a seed node randomly and then randomly a peer from the reported peers until we have 10 connections.

If a peer gets disconnected it will be removed from the connected peers and reported peers and a new connection will be opened to a new random peer.
Seed nodes have a higher max. connections limit (50).

Connection maintenance

We check in a random interval (5 min. - 10 min.) if the number of our connections exceeds the limit of 12 connections. If so we shut down the oldest connections.
We check the activity timestamp and if older than 30 sec. we send a PingMessage with a nonce. The receiving peer responds with a PongMessage and the received nonce. When receiving the PongMessage the peer checks if the nonce is the same, otherwise it disconnects the peer.

We also send periodically (1-2 min.) a GetPeersRequest to all connected peers to get the peersAddresses updated.

Both maintenance tasks are using a randomized delay (5-10 min).

With the GetPeersRequest the node is advertising himself to the network periodically as he sends his own address along with his peer addresses to his connected peers.

Data storage

Data is stored in a hash map with the SHA-256 hash of the data as key.
There are *add* and *remove* operations.
Removal of protected data is only accepted if the signature included in the message is valid.
The protected data holds the public key which was passed in the *add* message.
We use DSA 1024 for the signing of stored data and SHA-256 for the hash of the data and the sequence number.

There can be repeated *add* and *remove* messages and the order how they arrive at other peers could get mixed up. To get a correct ordering we use a sequence number which is increased at every new *add* or *remove* operation by the data owner. He needs to sign the hash of the data and the sequence number and the receiver verifies the signature as well as that the sequence number is larger as the already stored data (if already in the data map).
We use a map (data hash is key, sequence number value) which gets persisted to have a lookup for verifying the sequence numbers. So even in case we get a *remove* message first (we don't have the associated data) and get the *add* message later (with a lower sequence number as it was originally the first message) we can avoid to do an incorrect *add* operation.

Storage data protection

Add operation:

1. First we check that the public keys associated with the data are correctly matching. There are different cases for MailboxMessages and normal messages.
2. Next we check if the signature of the hash of the sequence number and the data is valid using the public key attached to the data. If valid we continue.

3. We check if we have the sequence number in our map and if so if the new sequence number is larger than the stored one. That is needed also if the data is not in our map (anymore) in case it was removed by the TTL check.
4. Next if we overwrite an existing data object we check if the public key of the stored data matches the public key of the new data.
5. If we did not have the data already we will broadcast it to our authenticated peers.

The **remove operation** is similar but we check initially if we have the data stored locally, if not we return.

The mailbox storage needs a bit of extra handling as the sender is the data owner when he adds the data to the network, but the receiver will be the one who has the permission to remove the data after he has applied it. So we use other data types and add the receiver's public key to the data.

Private data protection

Private data are protected as well by hybrid encryption (asym: RSA, 2048 and AES 128 for sym. encr.) and the sender's signature (SHA1withDSA).

Time to live

All stored data are expirable. Each node checks periodically every 10 minutes for expired data and removes that from its data map. When an object is received for storage the date is set to the current data. The TTL value is defined by object type.
The moment when an object becomes expired will not be synchronous in the network and we could get add messages after the object has expired. To avoid repeated additions of an already expired object when we get it sent from new peers, we don't remove the sequence number from the map. That way an *add* message for an already expired data will fail because the sequence number is equal and not larger.

Data re-publishing

When re-publishing data and therefore updating the TTL the data owner increments the sequence number.

Data types used for storage

Arbitrator: public data; TTL= 10 days (arbitrator re-publish every 5 days and at startup)
Offer: public data; TTL= 10 minutes (offerer re-publish every 8 min. and at startup)
Mailbox message: private data, encrypted and signed; TTL= 10 days
Alert: public, signed with developer key; TTL= 10 days

Data size

Data will be compressed.

Normal messages are all quite small. The initial GetData response will be one of the largest (up to a few MB uncompressed). The offer has about 2 KB. If we assume 100 - 1000 offer we get about 200 KB - 2MB.
The mailbox messages are hard to estimate but should not become too large as the trade messages are small and will be removed as soon the receiver is online.
In the dispute process the user can send attachments. We need to limit those to low MB values (5MB?).
Alert and Arbitrators data are small as well.
We estimate the total data set size in the range of 1-10 MB.

Flooding algorithm

Data are only broadcasted if we don't have it already in our data map, so preventing infinite broadcasts. Though we will receive the same data multiple times because of the flooding algorithm. When broadcasting we don't send the data to the peer from which we have received it.
Broadcast does not use any randomized delays and is executed in sequence.

Direct connections

Each direct communication uses encrypted and signed messages.
We use a hybrid encryption (asym: RSA, 2048 and AES 128 for sym. encr.) and signature (SHA1withDSA).

Seed nodes

Seed nodes run on hosted services and can count with more resources than normal network nodes. The limitation of 8 connections will be relaxed so that already connected nodes don't get disconnected when more nodes are bootstrapping. To actual limit needs to be investigated and can be set by an optional program argument. We use 50 connections at the moment as default.
The user can add a custom seed node via a command line argument. If that is set the default hard coded seed nodes will be ignored.

Attacks

Spam/DDoS

A user could misuse the network resources by sending spam messages.

Protection

- Authentication is used as protection against Sybil attacks
- First connection and data request is to a seed node
- Size limit for messages (both compressed and decompressed)
- Data type is checked

- The network ID is checked
- Violations lead to disconnection

Not implemented yet:

- The allowed frequency of messages from a peer is limited (aka “message throttling”)
- To protect against eclipse attacks we never replace more than 6 self initiated connections (authenticated inbound) with connection attempts requested from other nodes.
- We can store the addresses of misbehaving nodes and use ban scores for local blacklists
- Every node adds themselves to the shared data map with a short living TTL and republishing at half TTL time. That way we would get an independent live node list which cannot get easily attacked. At least the attacker would need to stay online with his hidden service and when he is attacking he gets disconnected quickly so he cannot build up a “connection reputation”.
- Treat reported peerAddresses from self initiated connections with a higher priority than those from connections from other peers. Should make eclipse attacks harder.
- Use seed nodes as anchor similar to https://eprint.iacr.org/2015/263.pdf

Not yet considered ideas:

- Bitcoin bonds could be used as additional Sybil protection
- PoW
- Reputation system could be introduced if needed
- Global blacklist could be introduced if needed

Simulations

- See how fast and reliable data is spread in a large network
- See how much message overhead is created by flooding the network
- Find best numbers for nr. of connected peers
- Check if random connection changes help
- Check if network splits or isolated islands can happen
- Try to simulate all kind of network level attacks and see how well the protection works

Bitcoins P2P network

Bitcoin uses a complicated mechanism for maintaining nodes.
We try to keep it as simple as possible and add complexity when problems are observed in network simulations. We have also other constraints and requirements as Bitcoin (much less scalability requirements,...).

References regarding Bitcoin P2P network

https://en.bitcoin.it/wiki/Satoshi_Client_Node_Discovery
https://en.bitcoin.it/wiki/Satoshi_Client_Node_Connectivity
https://cs.umd.edu/projects/coinscope/coinscope.pdf
https://github.com/bitcoin/bitcoin/blob/master/src/net.cpp

https://eprint.iacr.org/2015/263.pdf
http://fc14.ifca.ai/bitcoin/papers/bitcoin14_submission_17.pdf
http://arxiv.org/pdf/1405.7418.pdf

Open questions

- Versioning: We use a network and local data storage version in the serialization. Should we use a list of supported versions to be more flexible?
- The authentication handshake is done sequentially. Doing it in parallel could be considered but questionable if it improves performance. Total number of connections is then harder to check. Also during the startup process there is not expected much user activity (create offer) therefore a faster and more dense connection to the network is not essential.
- We could send only the offers and maybe only the offers for the users requested currency at the initial GetData request to keep data size smaller and reduce resource costs in case of DDoS attacks.
- We could split the functions for GetData and Authentication to separate seed node applications to gain more flexibility in deployment and DDoS protection. We can split the connected node and the GetData interface to be able to run a huge amount of lightweight GetData nodes (not connected to the P2P network but reading from disc the data which are written by the firewalled seed node). A pure GetData node can also offer the data to clearnet via HTTP servers.
- We could use the P2P storage for keeping track of well connected nodes and therefore creating kind of reputation. Authenticated nodes can be signed by the node which did the authentication and it could build up a kind of WoT-like hierarchy. Seed nodes have the highest trust level. The distance from the seed node can be used as weight for the reputation. That can be used to mitigate DDoS attacks at startup (GetData or AuthenticationRequest to wrong onion addresses). For new nodes which never got authenticated we cannot use that protection, but it would get limited to a smaller set, where we could use more strict limitations (special nodes, limited accepted connections per time interval,...)
- Checkout the DDoS protection built in Tor. Check if Exit nodes offer additional protection against DDoS, if so we can run seed nodes as clearnet servers to get that additional protection.
- Broadcast has no randomized timing behaviour added. Does this open up there any risks? If we do broadcast in parallel can it cause any performance peaks?
- PeerAddress updates: If a node goes offline it only gets removed from its connected peers neighbor set not from their neighbors neighbor set. Should we broadcast disconnects to update the actual live neighbor set faster? Should we also broadcast new neighbors to the complete network instead that it is just added locally? Would generate probably too much traffic. Dead nodes slow down bootstrapping but that is not critical if there are not too many and we use first a seed node. Size of set could be limited and time stamp could be introduced to be able to sort nodes which have been seen online recently. Problem of global time as timestamp would be set by any peer.
- Should we change connected neighbors after a certain time?

- Ban score policy. Which violations can be caused without bad intention?
- Could we use a fire-back strategy to invert DDoS attacks? If an attacker node send a message to 10 nodes and get back 10 message by each node he gets burnt harder than the network nodes. Though disconnection and local blacklists are probably a better strategy.
- Should we store locally the neighbor set for later startups? That would require timestamps as well, otherwise there is too less information about probability of the nodes availability.
- Which risks it would include to let not trusted persons operate a seed node. Basic requirement is that they are well connected and reliable. Fast software updates are another important factor.
- Java serialisation might not be the best solution for the protocol format. To change that to JSON or Protocol Buffers is for the moment too much effort but should be considered before going live to mainnet as a later change will be much harder.