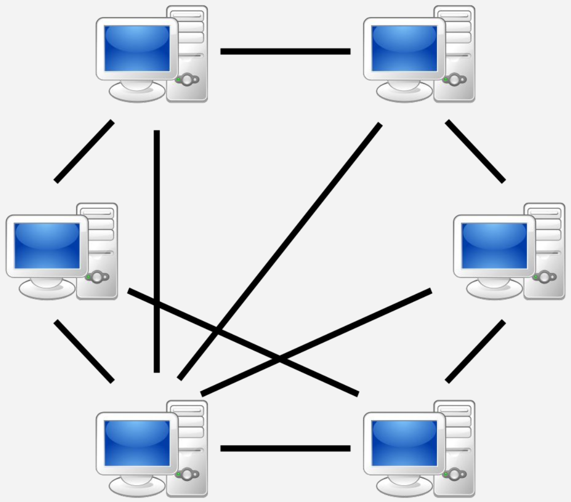
chaincode

P2P Design in Bitcoin Core

Reflections of a code contributor

What is the P2P layer?



1. Peer management (who do we connect to?)
2. Communication protocol/logic

Who do we expect to use the software?

1. Wallet users
2. Miners
3. Anyone who wants to validate

We assume that this software may be run by everyone.

Goals of the P2P layer

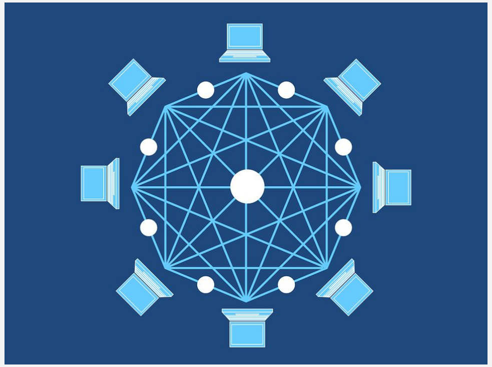
- Stay in consensus:
  - Connectivity to the honest network
  - Be able to download blocks and transactions of interest (mining, fee estimation, relay to miners, etc)
- But don't sacrifice:
  - Privacy
  - Efficient resource usage
  - Robustness to broken or malicious peers

What we’ll cover in this talk

- Problems we try to solve
- Examples of design choices we’ve made to solve those problems
- Some ideas for future work (definitely non-exhaustive!)

Disclaimer: There is a big design space here, and lots of potential ways to solve these problems!

Connectivity to the honest network



- Need at least 1 honest peer
- Can't force others to connect to us
  - New nodes have hardcoded peers and can use "DNS seeds"
  - Nodes gossip IP addresses of other nodes via "addr" messages.
    - (Data structure called "addrman" organizes info)
  - Use heuristics to find a diverse group of peers from addrman
  - Care must be taken so that "addr" message / processing can't be gamed, e.g. so that an attacker can take over the addrman!
  - Protect against Eclipse Attacks (Heilman, et al).
- Once we have peers, disconnect others that appear on an invalid chain. (risky?)

Downloading blocks and transactions

- We want to download blocks and transactions that are *of interest*:
  - For txs, it's everything that we would accept to our mempool.
  - For blocks, it's everything that we need to be on the same tip as everyone else.
- Generally, we request any transaction anyone announces to us (via an "INV"), and then decide if want it. We request any block that might help us advance our tip

Downloading blocks and transactions (2)

- Fundamental tradeoff between resource minimization and robustness to peer misbehavior.
- Much of the complication of our design is because we want to mitigate various DoS or privacy vulnerabilities. (Some things like compact blocks are complicated because they’re fancy.)

Considerations and tradeoffs

Lots of ways to achieve those two goals. But we're constrained in multiple ways:

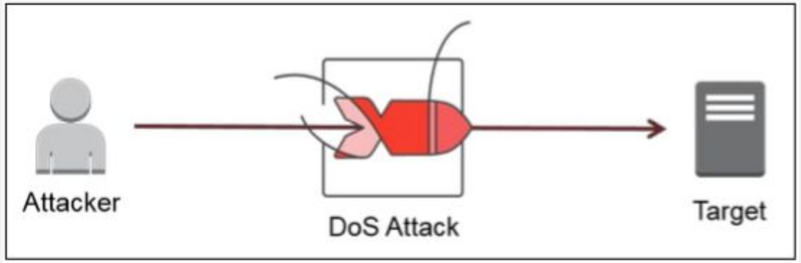
- Design complexity.
  - Simpler is usually better -- easier to implement/maintain/explain/analyze.
- Robustness to adversaries, misbehaving peers, network issues, etc.
  - Must eventually get all transactions, and ideally we should get all blocks quickly and efficiently.
- Resource utilization - ideally we don't waste too much bandwidth/memory/cpu/disk
- Privacy - we should achieve all these goals without leaking unintended information.

Need holistic reasoning when coming up with design.

Privacy goals

- Don’t let a node be fingerprinted by its application behavior (nodes should be identity-less).
- Don’t leak the originator of a transaction (protect user privacy for everyone).
- Don’t collect information about users of our software (no “phone-home” features).

DoS resistance: a grab-bag of problems



Non-exhaustive list of DoS issues that can arise:

- Fill up the disk attacks (can’t accept unnecessary blocks -- ties into validation layer)
- Network bandwidth attacks (“free relay”)
- CPU/Memory/Disk DoS
- Network partition attacks (i.e. no access to blocks)
- Transaction-relay DoS (eg InvBlock)

Efficiency and robustness

Many DoS issues overlap with (or indistinguishable from) robustness / efficiency issues.

- A DoS'er might be quick to announce a new block, but slow to deliver it (to slow down block relay). But so could a node on a Raspberry Pi that is HB-announcing compact blocks.
- Ideal behavior: never download data we don't need, and only download data once if we do need it, and get everything super fast.

In practice this is hard: peers might misbehave, go offline, etc. Can't wait forever for a peer to deliver data we need.

How do we think about all this?

Make intentional DoS attacks costly, in proportion to the magnitude of the attack.

- Requiring proof-of-work or tx fees to be paid can make attacks scale up in cost.

For robustness and efficiency, we have to be careful to achieve our performance goals without overly harming nodes on old hardware (the "unintentional" DoS-er), or even running old software.

- Example: what happens if everyone disconnects all nodes relaying an invalid block, after a soft-fork is deployed?

Case study: (legacy) block relay

- Old Block relay: send INV when you learned of a new block (just block hash, not full header).
- Nodes would immediately fetch a block upon receipt of block INV.
- When block arrives, process and store to disk as long as the block satisfied basic context-free checks (e.g. valid proof-of-work, valid merkle tree)
- DoS vector: low-work blocks could be announced to a node and be used to fill up the disk.
- *Solution?* Use proof-of-work as anti-DoS measure.
  - Require block header before deciding to download any blocks.
  - Announce blocks via header instead of using INV.
  - Only download blocks that lead to a more-work tip.
  - Don't process unrequested blocks.

Case study: connectivity to honest network

- Long time behavior: make 8 outbound connections, hope 1 is honest & connects us to honest network
- Sometimes we might know if we're at risk of being partitioned from honest network
  - Receive an invalid block / block header → peer who gave it to us is bad (disconnect?)
  - Peer stays on a less work chain than ours for an extended period of time → peer is bad (disconnect?)
  - No new blocks for an extended time period → maybe we're eclipsed by bad peers?

Case study: Handling risks of connectivity to honest network

- How might we handle risk of being partitioned? A few ideas:
    - General peer rotation (downside: adversary with a bunch of nodes will eventually eclipse)
    - Limited peer rotation -- try new peer occasionally to see if we learn something new
    - More outbound peers (tradeoff with resource utilization)

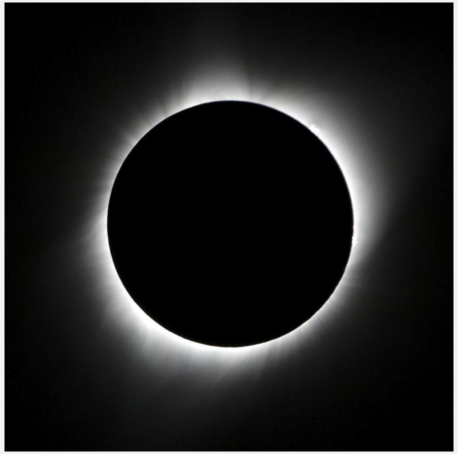
Case study: transaction relay (1)

- Long time behavior: peers send an INV for new transactions, nodes request all such txs from those peers
- Want to be resource efficient, don’t download the same tx multiple times
- Adversaries might like to know at which IP a tx first appeared on the network

Case study: transaction relay (2)

- What are possible ways to hide the network graph, to prevent attackers from deanonymizing?
  - Rapid peer rotation
  - Eliminate cross-peer optimizations
  - Recent proposal: separate the block and transaction networks
- Also should look to ways to augment/replace poisson relay with something that is less leaky

Future work: reduce partitioning risk



How to prevent eclipse attacks from moderately well-funded adversaries?

Imagine that an attacker has X% of the listening nodes on the network.

- What are the most effective attacks?
- How expensive to carry out such attacks?

Reducing bandwidth for tx relay

- *Basic problem*: current transaction relay system scales badly as more peers are added
- Send INV's for every tx along every link of the network
- Adding more connectivity to the network makes Bitcoin's network more robust (e.g. to eclipse attacks), but at the cost of more bandwidth

(Recent proposal, *Erlay*, to use a set reconciliation technique to replace INVs)

Block relay

Compact blocks (BIP 152) are awesome!

Yet many improvements are possible, such as:

- Maximize the likelihood of reconstruction with extra pool or prefilled transactions (or protocol extension to do even better)
- Parallel fetching of compact blocks (so that we're less dependent on the original announcer of a block if they become slow to finish relay)
- Parallel processing of network messages (so that we can respond with a BLOCKTXN message even while busy validating a block)

“Package relay”

This is a problem that spans the P2P layer and the validation layer.

Mempool acceptance (validation) has some anti-DoS quirks:

- Uses tx feerate to prioritize what gets into the mempool
- Txs also can only be added to mempool if all unconfirmed parents are in mempool (what would happen otherwise?)
- What if a very high-fee child tx depends on a very low-fee parent?

Tx “packages” are txs that have some kind of dependence relationship (jargon of our code).

Improving this requires work at both the validation layer and the p2p layer.

Summary

- P2P design is result of considering trade offs across many different goals/perspectives.
- Many design choices made as a result of firefighting
- Difficult to do wholesale rewrites of anything
- Lack systematic frameworks for measuring performance or evaluating problems.
- Think adversarially!