The Fault in our Lightning!

A study of Lightning Network's resilience against fault injections

Report for CS839 (Spring 2023) by Aditya Jain at UW—Madison

Introduction

Scalability is a critical concern in the Bitcoin blockchain. Low transaction throughput and high fees during times of congestion make the network practically unfeasible for micro-payments. Micropayment channels [1] are a solution to enable fast, low-cost transactions between two parties on a blockchain without actually having to record every transaction on the blockchain.

The Lightning Network [2] is a network of these channels that allows payments to be routed across multiple channels, enabling fast and scalable transactions. It significantly improves transaction speed and scalability, making it a promising solution for blockchain-based micropayments. At the time of writing, there are about 73k active channels and 16k active nodes. Lightning Network has been a wildly successful project with promising technology at its heart.

Given the size and complexity of the project, unexpected faults can have catastrophic outcomes on the network. Our motivation is a similar study on storage systems [3]. The key idea is to introduce faults, understand how they propagate through the network, and if they cause any unexpected failures. It is worth pointing out that there have been studies on attacking the lightning network [4], but our methodology is not to actively attack but rather study the behaviour of faults.

Overall in this project, we make the following contributions. First, we deploy our own private Lightning Network running over a private Bitcoin Network on CloudLab [5]. The node implementations we use are Lightning Network Daemon (LND) [6] and BTCD [7]. Next, we develop an RPC-based framework to interact with the lightning nodes. This allows us to automate the setup process and perform experiments quickly on many topologies. Finally, we consider two kinds of faults, Graph Faults and Invoice Faults, and analyze their behaviour.

Note that this report assumes that the reader understands the basics of payment channels and does not go into the details of channels/HTLCs and their design.Tour of the LND Codebase

LND is one of the most widely used full Lightning Network node implementations. The codebase is based on GoLang and has excellent developer friendly interfaces to support application development on top of LND. While there is no particular definition of what exactly is a component, the attempt here is to make sense of the codebase by breaking it down into logical components.

Networking

Brontide and LNwire make up the core of the networking setup in LND. Brontide is a secure crypto messaging protocol based on the Noise Protocol Framework. The implementation adheres to the lightning specification's BOLT #8 [8]. It ensures the cryptographic integrity of data being sent or received while also performing encryption and decryption. LNwire can be considered the transport protocol and determines the message exchange rules and format. By standardizing the message formats and communication rules, interoperability between different implementations of the Lightning Network is easily achieved.

The above setup usually runs on top of TCP, but this is not a hard requirement. Other transport layer protocols can also be used.

Graph

For each node, the knowledge of all (or most) active channels in the network, and their capacities is essential to route multi-hop payments efficiently. All nodes learn the complete graph of the network through gossip. The gossiped information includes the size, endpoints, and forwarding policy (e.g. fee) of a channel and a pointer to the output on the chain so that the shared information can be verified to avoid spam. We also note that when a new node joins the network, it fetches the network graph from the peers it connects with.

The graph is maintained in memory and stored in ChannelDB which uses BoltDB as the backend by default. Other backend database options like SQLite are also supported. A graph cache stores a subset of the network graph in memory. A channel cache stores the current state of channels a given node has open. In addition to these caches, a reject cache remembers the node and channel announcements that have been rejected to avoid wastage of resources in case these messages arrive again.

There are potential scalability limitations, but the LND dev community is confident that the issues will not be in the near future. Another related issue is with light nodes, e.g. mobile wallets. The solution for these nodes is to outsource the routing to designated Trampoline nodes. Payments are sent to these Trampoline nodes, which then figure out routing.

Routing

The complete route is determined at the sending node whenever a payment needs to be sent. A modified Dijkstra's algorithm is run, considering the nodes' reputation and fees and channels' capacities. Additionally, large payments can be split into smaller amounts (segments) that can be independently routed. This is achieved via the MPP (Multi-Path Payment) and AMP (Atomic Multi-Path Payment) mechanisms. The critical priority while creating segments is to have as few segments as possible, all of them roughly the same size.

It is worth noting that the entire route is determined at the payment sender node. Intermediate nodes only know about the immediate next node they need to route the payment to. Thus comes the term "Onion Routing" since every intermediate node peals a layer, figures out the next node, and so on. This scheme is privacy-preserving.

RPC Interfaces

There are two exported primary RPC interfaces: an HTTP REST API and a gRPC service (called LNRPC). In our framework we make use of LNRPC to automate the setup and test process.

Payments and Invoices

Payments in the Lightning Network are typically done via invoices. The recipient generates an invoice containing payment metadata and recipient information. Bolt 11 [9] provides the complete specification.

Upon generation, the invoice is stored in the InvoiceDB. Note that at the time of writing InvoiceDB is physically located in ChannelDB, although at some point it may be moved. It is a key-value store that maps payment request with the corresponding invoice metadata. As we will later find, it is absolutely critical for the recipient to store this information correctly in the store to avoid issues.

The recipient can share the invoice with the sender who can then make the payment using the information contained in the invoice.

Clearly, invoices are a handy and easy way to manage payments. However, they are not the only way. Spontaneous payments are when the payer initiates the payment and no invoice is needed. This is exposed via the *keysend* payment feature. Often, keysend payments are preferred for micropayments and when the amount is not fixed beforehand.

Autopilot

Autopilot is one of the vital non-mandatory components of LND. Non-mandatory because a sophisticated user might not use it. However, most LND nodes are expected to be running on autopilot. This feature provides automatic channel creation and routing.

For automatic channel creation, autopilot must solve a complex problem — figuring out which peers are most suited to open channels with. For this, the algorithm uses information from the network graph and connects with top-k (by default k=5) nodes with the highest Betweenness Centrality [10]. Intuitively this means that the nodes that are most central to the network, and so most critical, are preferred. Consequently, autopilot increases the overall centralization in the network as the same set of nodes keeps becoming increasingly crucial for the network over time. However, for the client, this way, the autopilot can optimize the fees for most payments.

Test and Setup Framework

As part of this project, we have developed a framework called SoyMocha [11]. It is highly automated for quick setup and experimentation. Currently it supports the setup of Lightning Network and Ethereum. We plan to bring more blockchains in its scope as we continue with the work.

Detailed instructions on usage are provided in the SoyMocha repository. Starting with the CloudLab manifest, our tools generate scripts to connect with all nodes via TMUX. Next, they copy configuration files and install necessary applications on the nodes. Finally, BTCD is started on one of the nodes, while LND on all others. For our work, a single Bitcoin node suffices.

SoyMocha also provides tools for graph management. Starting from a graph configuration file as shown below, our tools are able to create arbitrary topologies as specified. The simplicity of adjacency list representation adds to the power of our tooling.

```
graph.conf
edge=2,3,20000
edge=2,4,20000
edge=3,5,20000
```

Experiments

In this work, two faults are experimented with and discussed. The key intuition is to understand what happens when the software assumes a piece of data has been stored to a database successfully while in reality it is either not written at all or has been corrupted. The experiments reveal interesting side effects of such faults.

Graph Faults

Payment routing in the Lightning Network requires correct and up-to-date knowledge of the channel graph. Channel and node announcements are broadcasted to the network via gossip. Nodes in the network maintain the graph in their persistent store. When a new node joins the network it fetches the graph from its peers and then listens to the channel and node announcements to keep the graph information updated.

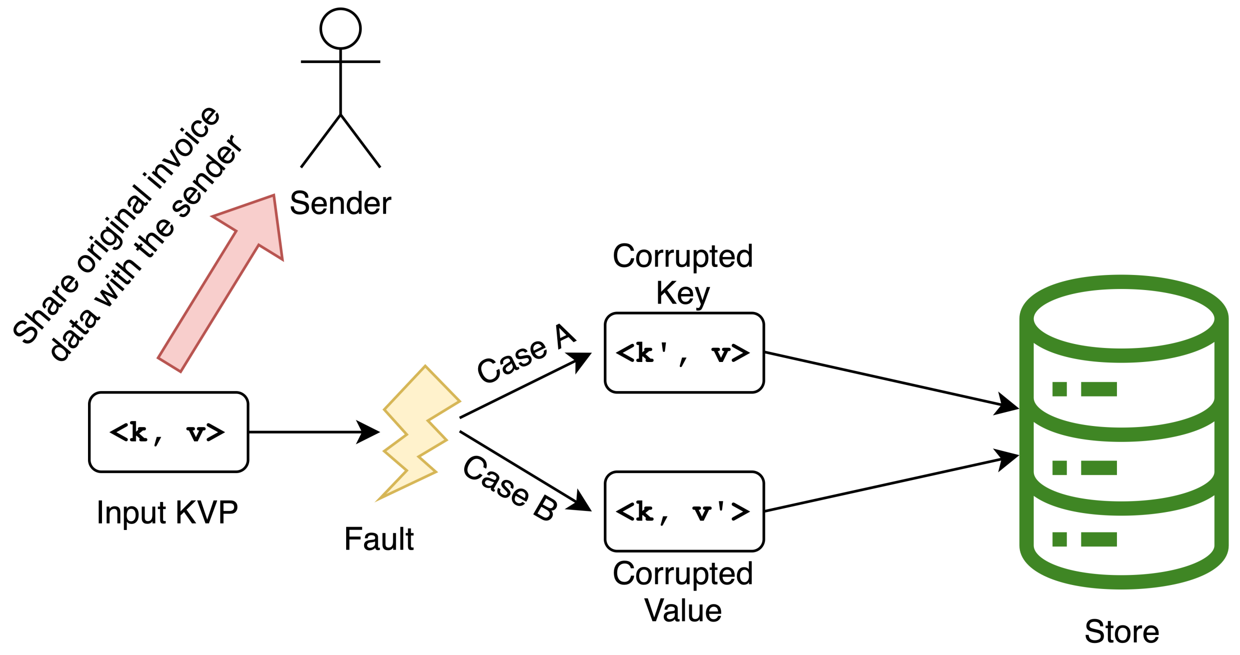
The first experimental finding is as follows. When a new node joins the network, it queries the network graph from the first peer it connects with. Subsequent connection establishment with other peers does not trigger graph fetch. This observation is crucial for this fault injection to work. Moreover, it is also counter intuitive as to why no attempt is made to fetch graph when more peers are added.

To understand the implications of the above finding, we can consider the following fault injection. We introduce an *Faulty Node (FN)* to the network. This node stops maintaining the network's graph in its store while keeping everything else functioning as usual. Such a fault can happen if for some reason the ChannelDB store has some errors.

Now we consider a new node, the *Victim Node (VN)*, and understand what happens when it joins the network. Let us assume that the first peer that VN connects with is FN. Clearly, since FN does not maintain the network graph, it returns an empty graph to VN. Furthermore, we know that any further connections that VN may establish with other peers will not yield the graph either. Consequently, VN is stuck with an empty initialized graph and needs to rely on gossip to build up the graph from scratch.

This propagation of fault can cause significant distress to the VN.

Invoice Faults



In Lightning Network, a receiver generates an invoice which can be used by the sender to make the payment. Invoices are cryptographically signed for security purposes. Invoices, at their generation time, get stored in the InvoiceDB as a key-value pair (KVP), where the key is the payment hash and the value is the corresponding invoice metadata.

Faults can be introduced in the manner as shown in the figure. Specifically, when a KVP $\langle k, v \rangle$ is inserted to the InvoiceDB, some corruption is introduced which can either corrupt the key or the value.

Case A is when the key (payment hash) is corrupted and instead of $\langle k, v \rangle$ the database stores $\langle k', v \rangle$ where $k' \neq k$. Consequently if the application queries for key k, the database fails to find it manifesting this fault as a missing key. In Case B, the value (invoice metadata) is corrupted instead and instead of $\langle k, v \rangle$ the database stores $\langle k, v' \rangle$ where $v' \neq v$. When the invoice is eventually used, corrupted data is observed. In our experiments, we specifically focussed on Case A faults as they were simple to reason about and yet led to interesting behavior. Once again, we introduce a *Faulty Node (FN)* to the network which injects Case A faults when writing invoice data to InvoiceDB.

It must be noted that in our fault injection, while the data being stored to InvoiceDB is faulty, the in-memory state of the application is correct. Furthermore, any logs and outputs printed at invoice generation are also correct. This implies that recipient (FN) is able to share non-faulty original invoice I_F with the sender S. It is just that the FN's store holds corrupted invoice data which causes faults when the invoice is eventually used by the sender to send out the payment. For simplicity, in our experiments we set up a direct channel between S and FN. The following are the observations.

1. We note that the act of storing faulty invoice data itself causes no visible harm to FN and it continues to operate normally.
2. Normal operation implies two things. First, it is able to receive payments on non-faulty invoices that it has correctly stored in InvoiceDB. And second, any payments being routed through FN continue to happen without issues.
3. Let us now consider that now sender S decides to use I_F to make a payment to FN. We note that I_F is a completely valid and cryptographically signed invoice and so S has no suspicions about using it. We find that this payment hangs. S continues to observe as the payment being "in-progress" until the channel eventually gets closed. In the meantime, FN is considered to have entered Faulty Operation.
4. During faulty operation the following is observed. First, FN is unable to receive payments on even the non-faulty invoices that are correctly stored in InvoiceDB. In fact, all these payments are also found to hang. And second, payment routing through FN also fails.
5. We note that the channel between S and FN is gets closed after a few minutes.

It is interesting that such a small scale fault injection can cause significant fault in the network and affect a large number of payments.

Summary

The following table summarizes the discussion for the two experiments.

Fault	Where	Description	Result
Graph	Faulty Node	A faulty node fails to store network channel graph in its ChannelDB store.	Victim Nodes end-up with an empty graph store and rely on gossip to slowly discover the network based on the activity. Payments at victim nodes may be affected.
Invoice	Faulty Node	A faulty node which intends to receive a payment generates an invoice and shares with a sender. However, corruption happens when the invoice is stored to the faulty node's InvoiceDB store.	Normal operation until the sender attempts a payment to the incorrectly stored invoice. Faulty operation afterwards in which all payments (to the faulty node as well as those routed through it) fail.

Conclusion

Lightning Network is in an exciting phase of development and has been an enormous success. This study has been an important first step towards understanding the LND code base in detail. Furthermore, we attempt to address the need to test Lightning Network implementations' resilience against unexpected software faults. Our experiments show that small faults have the potential to turn into more significant failures causing multiple payment failures and channel closures, resulting in both monetary loss and computational churn in the network.

Future Work

This study is in no way complete and is a mere starting point for a variety of experiments that can be and should be done to further expand our understanding. One of the major limitations of this work is the focus on a single implementation LND. All the tools and software developed for this project are tightly coupled with LND. As the logical next step, all the experimentation should be extended to include other implementations like c-lightning, Eclair, and LNP. Inclusion of other implementations opens up an exciting dimension of studying how faults propagate through different implementations when they work together in the network. Additionally, there are some immediate next steps for the experiments. For example, the role of the autopilot needs to be taken into consideration while estimating the impact of graph faults. Similarly, case B needs to be studied for invoice faults.

Code Location

The setup and testing framework is located here: https://github.com/ajain365/soymocha

References

1. Decker, Christian, and Roger Wattenhofer. "A fast and scalable payment network with bitcoin duplex micropayment channels." In Stabilization, Safety, and Security of Distributed Systems: 17th International Symposium, SSS 2015, Edmonton, AB, Canada, August 18-21, 2015, Proceedings 17, pp. 3-18. Springer International Publishing, 2015.
2. Poon, Joseph, and Thaddeus Dryja. “The Bitcoin Lightning Network: Scalable Off-Chain Instant Payments." Version 0.5. Publication or Organization, 2016. Retrieved from https://lightning.network/lightning-network-paper.pdf.
3. Ganesan, Aishwarya, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions." In FAST, pp. 149-166. 2017.
4. Mizrahi, Ayelet, and Aviv Zohar. "Congestion attacks in payment channel networks." In Financial Cryptography and Data Security: 25th International Conference, FC 2021, Virtual Event, March 1–5, 2021, Revised Selected Papers, Part II, pp. 170-188. Berlin, Heidelberg: Springer Berlin Heidelberg, 2021.
5. Duplyakin, Dmitry, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller et al. "The Design and Operation of CloudLab." In USENIX Annual Technical Conference, pp. 1-14. 2019.
6. Lightning Labs. “API Guide.” Accessed May 12, 2023. LND Documentation. Lightning Labs. Retrieved from https://lightning.engineering/api-docs/api/lnd/
7. BTCD. Computer software. Version 0.23.3. btcsuite, 2022.
8. Lightning Network Specifications. "BOLT #8: Encrypted and Authenticated Transport." Version 1.0.0. 2020. Retrieved from https://github.com/lightning/bolts/blob/master/08-transport.md
9. Lightning Network Specifications. "BOLT #11: Invoice Protocol for Lightning Payments." Version 1.0.0. 2022. Retrieved from https://github.com/lightning/bolts/blob/master/11-payment-encoding.md
10. Freeman, Linton C. "A set of measures of centrality based on betweenness." Sociometry (1977): 35-41.
11. Soymocha. Computer software. Version 1.0.0. Aditya Jain and Hunter Goff, 2023.