Cypherpunk Cogitations

8 JANUARY 2018

The Challenges of Building Ethereum Infrastructure

Several of our engineers, myself included, have spent a substantial portion of the past year working on building low level Ethereum infrastructure at BitGo. We've encountered a plethora of challenges along the way; in this post we'll cover some of the more interesting ones.

EthereumJ Issues

Our first attempt (in 2016) at building a highly indexed database of the Ethereum blockchain used EthereumJ to run the EVM in the occurred on the network. This ended up being a poor decision. EthereumJ was not production quality in 2016; it may be better today but according to my conversations with Ethereum developers, it's still unreliable. We experienced multiple crashes, subpar documentation, and not so great developer support from the small number of EthereumJ experts. We lost count of how many times our EthereumJ client got stuck or corrupted and we had to delete the entire chainstate and re-sync from scratch. We *never managed to stabilize* our EthereumJ indexer to the point that we were confident it was ready for production use. Interestingly, I came across this StackExchange post that noted stability issues as of June 2017.

256 bit support

A fundamental problem we've had while dealing with Ethereum is how to handle 256 bit numbers. The design rationale for the Ethereum Virtual Machine states:

If you're hoping to store them in any popular production quality database as numeric types that you can query and manipulate as such, you're in for a surprise. We found that mongo only supports 128 bit numbers, but due to a limitation in mongoose (the NodeJS mongo driver) the actual support was 32 bits. Because our indexer has traditionally been multithreaded in order to increase performance, we really needed the ability to atomically increment and decrement balances in mongo while we were processing many transactions simultaneously. It's quite likely that we could be a simple "get & put" operation could overwrite data. As such, we ended up developing a new data structure we called the "BitGo BigInt" that supported 256 bit numbers by breaking the values apart and storing them into 6 separate database fields. A conversion library made it simple to switch between BigIntegers and BitGo BigIntegers.

Here you can observe the loss of precision when performing 256bit math operations with mongodb

After working on it for several months, the Ethereum network itself then suffered a series of attacks that caused a loss in confidence for the network as a whole. We shelved the Ethereum project in late 2016 despite the demand from our exchange customers, determined to revisit it when we were more confident that we would be able to deliver the quality our customers deserve.

If at first you don't succeed, try and try again

As we saw the general rise in crypto assets beginning to occur, it was clear that we needed to transition from just supporting Bitcoin to supporting any crypto asset with sufficient transaction volume to make it worth diversifying our revenue streams. As such, a rearchitecting of our infrastructure was required in order to make it more flexible for supporting a variety of blockchains. We began writing a third indexing service from scratch, informally named the "generic indexer." It was designed to be more modular and thus support speaking to a variety of full node RPC interfaces, with

This rearchitected indexer service made it immensely easier to add Ethereum support, though we still ran into a few challenges.

We wanted to completely avoid the hassle of using a custom data type to store huge numbers, so we settled upon storing values as strings instead. The downside is that we can't query them using numeric operators, but we've managed to avoid needing to do so by keeping this limitation in mind while developing the wallet. The other downside is that we can't atomically update the values—we must instead read the string from the database, convert it to a BigInteger, modify the value in memory, convert it back to a string, and then store it. In order to account for this design limitation, we have single threaded our indexing service for Ethereum. By using a few neat performance tricks we picked up along the way, our single threaded Ethereum indexer is actually orders of magnitude faster than our original multi-threaded EthereumJ-based indexer. More specifically—we no longer run the EVM in our indexer—we outsource that work to Parity nodes. We also decided not to index the entire Ethereum blockchain and to only index transactions related to BitGo wallets. By throwing away the vast majority of data that we pull from the node, we were able to switch from a multithreaded indexing service to a single threaded service while still gaining an order of magnitude speedup and decreasing our data storage requirements by several orders of magnitude.

Parity Issues

We chose to use Parity as our node the second time around given that it had the reputation of being the most robust and performant. This has worked a lot better than trying to run consensus code inside of the indexing service itself, but it hasn't been foolproof. It's not quite as robust as Bitcoin Core in our experience— we have experienced a variety of unexplained crashes and stalls.

> thread 'IO Worker #1' panicked at 'Can't create handshake:
> Auth', /buildslave/rust-buildbot/slave/stable-dist-rustc-linux/build/src/libcore/result.rs:837
> stack backtrace:
> 1: 0x7f97790acafa—<unknown>
> 2: 0x7f97790b3e7f—<unknown

Those crashes occurred with 1.6.3 and we have since upgraded to 1.7.2 which seems to have fixed that issue—however, this new version has issues with the initial sync in archive mode that causes it to completely blow out the disk usage and crash on a regular basis from filling up the disk. The only solution so far has been to use a cron script to restart the node every hour or so, which somehow cleans up all the excess data on disk. We're hopeful that this recent change will fix the issue.

We also continue to experience complete loss of all peer connectivity on a regular basis with our Kovan network Parity nodes; this results in our test environment nodes falling out of sync with the network. The only fix we've found for this is to delete the nodes.json config file and restart Parity. We're told it was an issue with the boot nodes that was recently fixed, but we've still been experiencing stalls since upgrading.

Running an archival Ethereum node requires a ton of disk I/O since you're executing every smart contract ever performed in the history of Ethereum and updating the state on disk. It's common knowledge now that if you try to sync an archival node on a machine with spinning disk drives, it will never catch up to the tip of the blockchain—there simply aren't enough IOPs available. Thus, you MUST have a solid state disk to run an Ethereum node. However, this gets tricky on virtualized servers where IOPs on reliable data stores are not cheap.

To give you a general idea of disk I/O performance.

We started syncing a 1.7.2 archival node on a VPS with 2,000 IOPs provisioned, traveled to Japan, and when I returned 4 days later it was still only at block 3,800,000 out of 4,300,000.

> *This week I learned you can start syncing an archival Ethereum node, travel completely around the world, and it still won't be finished.*
>
> — Jameson Lopp (@lopp) September 23, 2017

We then gave it another shot with a VPS that had 5,000 IOPs. It took 8 days to get to block 4,300,000. The further you progress along the blockchain, the harder the blocks become to verify due to the increased transaction volume. For example, we noted that it took over 4 minutes to verify this block (and many others) while the machine was pegged at a constant disk read of 50 MB/S and write of 50 MB/S. As of December 2017, the Ethereum network has become even more popular and our production Parity nodes sometimes struggle to stay synced with the tip of the chain due to the high disk I/O. One possible "solution" to this would be for us to switch to using ephemeral storage on AWS. This is a directly attached SSD that isn't metered or throttled. The drawback is that it's not guaranteed to be durable. Doing a "stop/start" on the instance will wipe out the data. Also, if the underlying drive fails, the data will be lost since it isn't mirrored.

At time of writing the chatter I'm hearing amongst other enterprise Ethereum-based services is that both geth and parity are suffering network. Hopefully the next releases of popular Ethereum node software will be focused on performance improvements.

BitGo's MultiSig Contract

Unlike Bitcoin and many other crypto assets, Ethereum does not have native multisignature functionality for securing funds. BitGo's business model requires that all crypto assets be secured in a 2-of-3 multisig wallet so that BitGo can act as an oracle / cosigner on transactions to enforce security policies. As a result, we had to implement multisig with a smart contract. You can check it out here.

Writing a smart contract to secure crypto assets is pretty dangerous; it's not far from "rolling your own crypto" which is one of the fundamental things to avoid in this industry. Smart contract security is a whole other can of worms; BitGo is dedicating a future blog post to covering the security issues we encountered and steps we took to resolve them.

Smart Contract Complexities

When parsing events emitted by smart contracts it can be tricky to determine just where the money came from. In UTXO-based systems such as Bitcoin, all you have to do is look at the transaction inputs. In account-based systems, on a standard transaction you can just look at the "from" account. However, when the value is being sent via smart contracts it becomes much more complicated. BitGo's multi-sig contract also has "forwarding address" functionality that adds more complexity to this logic.

Our first approach was to use EthereumJ to load our contract ABI and parse transaction receipts with it. It seemed to work well for parsing addresses out of the event log data, but it would

BigIntegers.) After many frustrating hours of experimenting and trying to get help on the EthereumJ gitter, we ended up once again ditching EthereumJ and ended up doing manual parsing of the byte arrays from our contract transaction event logs. It feels rather hacky to say "get the Nth 32 byte chunk of data out of this byte array" but at least it works.
Smart contract debugging is also quite challenging with the current state of developer tools. It is hard to know where a transaction failed - often you'll just get a vague error like "bad jump destination". Hopefully we'll have better tools to debug smart contracts in the future.

Network-Wide Lack of Smart Contract Support

Other cryptocurrencies have taught us some lessons with regard to send/receive compatibility across services, especially with regard to address format compatibility. For example, LTC implemented a new P2SH address format in order to make their addresses incompatible with BTC P2SH addresses. Prior to this, the address formats were the same which allowed users to accidentally send BTC to LTC addresses and vice versa. This causes headaches for services to recover the assets.

> *When we launched the first multi-sig BTC wallet, many sites rejected "3" addresses as invalid. Today, same problem with LTC "M" addresses.*
>
> — BitGo (@BitGo) June 19, 2017

/still/ don't correctly validate the new address format. This type of issue is inconvenient because it can cause users to not be able to send funds from one service/wallet to another because the sending service thinks the "to" address is invalid. However, Ethereum has a unique problem that enables users to send funds to a service but the service *never sees the deposit*. This is because many services take the easy route when implementing ETH support and they only listen for standard ETH transactions by inspecting the "to" and "from" address fields on the main transaction. However, if a transaction is executing a smart contract then the movement of funds by contract events that are fired do not get reflected in "to" and "from" address fields on the transaction. Rather, they are reflected by "internal transactions" that must be generated by executing the smart contract and parsing the event output. Because many services don't bother to inspect smart contract events to see if they are making deposits to addresses owned by the service, BitGo wallet users may inadvertently send ETH to a service and not get credited, causing confusion and requiring intervention by human support staff. We expect that eventually this problem will subside as smart contract sending becomes more common and services realize that listening for deposits from smart contracts is **not an optional "nice to have" feature**.

Nonce Issues

Unlike bitcoin, a nonce has to be included with every ethereum transaction. The nonces have to be sequential when sending from a particular address. Hence only nonce 5 can get confirmed after a nonce 4 transaction has been confirmed from a particular address. It is a challenge managing nonces for multiple wallets, particularly if a transaction fails to be broadcast or confirmed due to any number of reasons. We have built tools to manage this, but in general a transaction that fails to broadcast will block future unconfirmed transactions in UTXO based crypto assets. Wallet software needs to be able to handle failed or stuck sends so that they don't cause the entire wallet to become unusable.

Address Generation

Services that accept deposits from many users need to generate a unique address for each user so that they know which user's account to credit when they receive money. However, generating addresses for smart contracts is more complex than generating addresses for other blockchains. On other chains you can generate as many addresses offline as quickly as you want and if they are never used, it's no big deal. But with smart contracts, each newly generated address must be posted as an event to the blockchain. As such, it costs money, takes time, and is a blocking operation. Because the blockchain must know about the address before it can receive a deposit, it's possible to generate an address and display it to the user, who then sends funds to it. But if their transaction gets confirmed before the transaction that generates the address, the user's money won't be detected as a deposit into the smart contract and **will have to be manually recovered**! To prevent users from shooting themselves in the foot, we don't allow users to see newly created addresses until *after* the address generation transaction has been confirmed.

Network Backlogs

Backlogs on the Bitcoin network have been common for over a year and we have made adjustments to our fee algorithms to compensate. However, backlogs in Ethereum are relatively new and have side effects we didn't anticipate. The primary problem is with wallet initialization, which is similar to the address generation issue described previously. Because a transaction must be broadcast and confirmed in order to initialize a wallet contract, and the user begins sending other transactions to make deposits, it's not possible for the Ethereum network to fire the appropriate contract events because *the contract doesn't exist yet*! Thus we've had to add extra logic that locks the wallet and prevents users from seeing the wallet's deposit address until after the contract creation transaction has been confirmed.

The Road Ahead

The complexities in operating a smart contract based multisignature Ethereum wallet have surprised us on several fronts. We still have plenty of work to do in both the operational and user experience areas and we are eagerly observing the continued evolution of development tools and best practices for smart contract security. BitGo's next step with our Ethereum infrastructure will be support for ERC20 tokens; we expect that the system we have built will continue to undergo trial by fire as it has to handle higher loads and secure even larger amounts of value.

Jameson Lopp
Read more posts by this author.

Read More

Securing Your Financial Sovereignty

2017 is turning out to be the year of the airdropped Bitcoin fork. First Bitcoin

SegWit2x. Now the ecosystem…

Satoshi Roundtable IV Recap

It was my pleasure to once again attend the Satoshi Roundtable; this year it was scaled up immensely and I didn’t even have a chance to…

10 MIN READ

5 MIN READ