Base Layers And Functionality Escape Velocity

2019 Dec 26

See all posts

One common strand of thinking in blockchain land goes as follows: blockchains should be maximally simple, because they are a piece of infrastructure that is difficult to change and would lead to great harms if it breaks, and more complex functionality should be built on top, in the form of layer 2 protocols: state channels, Plasma, rollup, and so forth. Layer 2 should be the site of ongoing innovation, layer 1 should be the site of stability and maintenance, with large changes only in emergencies (eg. a one-time set of serious breaking changes to prevent the base protocol's cryptography from falling to quantum computers would be okay).

This kind of layer separation is a very nice idea, and in the long term I strongly support this idea. However, this kind of thinking misses an important point: while layer 1 cannot be *too* powerful, as greater power implies greater complexity and hence greater brittleness, layer 1 must also be *powerful enough* for the layer 2 protocols-on-top that people want to build to actually be possible in the first place. Once a layer 1 protocol has achieved a certain level of functionality, which I will term "functionality escape velocity", then yes, you can do everything else on top without further changing the base. But if layer 1 is not powerful enough, then you can talk about filling in the gap with layer 2 systems, but the reality is that there is no way to actually build those systems, without reintroducing a whole set of trust assumptions that the layer 1 was trying to get away from. This post will talk about some of what this minimal functionality that constitutes "functionality escape velocity" is.

A programming language

It must be possible to execute custom user-generated scripts on-chain. This programming language can be simple, and actually does not need to be high-performance, but it needs to at least have the level of functionality required to be able to verify arbitrary things that might need to be verified. This is important because the layer 2 protocols that are going to be built on top need to have some kind of verification logic, and this verification logic must be executed by the blockchain somehow.

You may have heard of Turing completeness; the "layman's intuition" for the term being that if a programming language is Turing complete then it can do anything that a computer theoretically could do. Any program in one Turing-complete language can be translated into an equivalent program in any other Turing-complete language. However, it turns out that we only need something slightly lighter: it's okay to restrict to programs without loops, or programs which are guaranteed to terminate in a specific number of steps.

Rich Statefulness

It doesn't just matter that a programming language *exists*, it also matters precisely how that programming language is integrated into the blockchain. Among the more constricted ways that a language could be integrated is if it is used for pure transaction verification: when you send coins to some address, that address represents a computer program `P` which would be used to verify a transaction that sends coins *from* that address. That is, if you send a transaction whose hash is `h`, then you would supply a signature `S`, and the blockchain would run `P(h, S)`, and if that outputs TRUE then the transaction is valid. Often, `P` is a verifier for a cryptographic signature scheme, but it could do more complex operations. Note particularly that in this model `P` *does not have access* to the destination of the transaction.

However, this "pure function" approach is not enough. This is because this pure function-based approach is not powerful enough to implement many kinds of layer 2 protocols that people actually want to implement. It can do channels (and channel-based systems like the Lightning Network), but it cannot implement other scaling techniques with stronger properties, it cannot be used to bootstrap systems that do have more complicated notions of state, and so forth.

To give a simple example of what the pure function paradigm cannot do, consider a savings account with the following feature: there is a cryptographic key `k` which can initiate a withdrawal, and if a withdrawal is initiated, within the next 24 hours that same key `k` can cancel the withdrawal. If a withdrawal remains uncancelled within 24 hours, then anyone can "poke" the account to finalize that withdrawal. The goal is that if the key is stolen, the account holder can prevent the thief from withdrawing the funds. The thief could of course prevent the legitimate owner from getting the funds, but the attack would not be profitable for the thief and so they would probably not bother with it (see the original paper for an explanation of this technique).

Unfortunately this technique cannot be implemented with just pure functions. The problem is this: there needs to be some way to move coins from a "normal" state to an "awaiting withdrawal" state. But the program `P` does not have access to the destination! Hence, any transaction that could authorize moving the coins to an awaiting withdrawal state could also authorize just stealing those coins immediately; `P` can't tell the difference. The ability to change the state of coins, without completely setting them free, is important to many kinds of applications, including layer 2 protocols. Plasma itself fits into this "authorize, finalize, cancel" paradigm: an exit from Plasma must be approved, then there is a 7 day challenge period, and within that challenge period the exit could be cancelled if the right evidence is provided. Rollup also needs this property: coins inside a rollup must be controlled by a program that keeps track of a state root `R`, and changes from `R` to `R'` if some verifier `P(R, R', data)` returns TRUE - but it only changes the state to `R'` in that case, it does not set the coins free.

This ability to authorize state changes without completely setting all coins in an account free, is what I mean by "rich statefulness". It can be implemented in many ways, some UTXO-based, but without it a blockchain is not powerful enough to implement most layer 2 protocols, without including trust assumptions (eg. a set of functionaries who are collectively trusted to execute those richly-stateful programs).

Note: yes, I know that if `P` has access to `h` then you can just include the destination address as part of `S` and check it against `h`, and restrict state changes that way. But it is possible to have a programming language that is too resource-limited or otherwise restricted to actually do this; and surprisingly this often actually is the case in blockchain scripting languages.

Sufficient data scalability and low latency

It turns out that plasma and channels, and other layer 2 protocols that are fully off-chain have some fundamental weaknesses that prevent them from fully replicating the capabilities of layer 1. I go into this in detail here; the summary is that these protocols need to have a way of adjudicating situations where some parties maliciously fail to provide data that they promised to provide, and because data publication is not globally verifiable (you don't know when data was published unless you already downloaded it yourself) these adjudication games are not game-theoretically stable. Channels and Plasma cleverly get around this instability by adding additional assumptions, particularly assuming that for every piece of state, there is a single actor that is interested in that state not being incorrectly modified (usually because it represents coins that they own) and so can be trusted to fight on its behalf. However, this is far from general-purpose; systems like Uniswap, for example, include a large "central" contract that is not owned by anyone, and so they cannot effectively be protected by this paradigm.

There is one way to get around this, which is layer 2 protocols that publish very small amounts of data on-chain, but do computation entirely off-chain. If data is guaranteed to be available, then computation being done off-chain is okay, because games for adjudicating who did computation correctly and who did it incorrectly *are* game-theoretically stable (or could be replaced entirely by SNARKs or STARKs). This is the logic behind ZK rollup and optimistic rollup. If a blockchain allows for the publication and guarantees the availability of a reasonably large amount of data, even if its capacity for *computation* remains very limited, then the blockchain can support these layer-2 protocols and achieve a high level of scalability *and* functionality.

Just how much data does the blockchain need to be able to process and guarantee? Well, it depends on what TPS you want. With a rollup, you can compress most activity to ~10-20 bytes per transaction, so 1 kB/sec gives you 50-100 TPS, 1 MB/sec gives you 50,000-100,000 TPS, and so forth. Fortunately, internet bandwidth continues to grow quickly, and does not seem to be slowing down the way Moore's law for computation is, so increasing scaling for data without increasing computational load is quite a viable path for blockchains to take!

Note also that it is not just data capacity that matters, it is also data latency (ie. having low block times). Layer 2 protocols like rollup (or for that matter Plasma) only give any guarantees of security when the data actually is published to chain; hence, the time it takes for data to be reliably included (ideally "finalized") on chain is the time that it takes between when Alice sends Bob a payment and Bob can be confident that this payment will be included. The block time of the base layer sets the latency for anything whose confirmation depends things being included in the base layer. This could be worked around with on-chain security deposits, aka "bonds", at the cost of high capital inefficiency, but such an approach is inherently imperfect because a malicious actor could trick an unlimited number of different people by sacrificing one deposit.

Conclusions

"Keep layer 1 simple, make up for it on layer 2" is NOT a universal answer to blockchain scalability and functionality problems, because it fails to take into account that layer 1 blockchains themselves must have a sufficient level of scalability and functionality for this "building on top" to actually be possible (unless your so-called "layer 2 protocols" are just trusted intermediaries). However, it is true that beyond a certain point, any layer 1 functionality *can* be replicated on layer 2, and in many cases it's a good idea to do this to improve upgradeability. Hence, we need layer 1 development in parallel with layer 2 development in the short term, and more focus on layer 2 in the long term.