The Great dApp Gap: Tunnelling over the barriers into mass dApp adoption

Introducing **Quantum Fusion Blockchain Network** *(QFN)*

This is the first introduction of the concepts underpinning the to-be-released Quantum Fusion Blockchain Network system, currently under development.
While the preliminary research and theoretical foundations of this paper should be *solid and sound*, the usual "other 80% of R&D work" still remains to be performed, so nothing written in the text below should be considered as final, not in theoretical computer science neither in the practical engineering sense.

qf.foundation team | September 2024

It's been 15 years since Bitcoin's "Chancellor on brink of second bailout for banks" launch moment. In this time, no blockchain-powered dApp became mainstream. Even you, dear blockchain-savvy reader, don't use any dApps daily. Meanwhile, Telegram, launched five years later, is nearing a billion users. Why is that, and what can be done to overturn the situation?

We don't believe it's due to a lack of talented engineers — the blockchain space has consistently attracted the best of the best for the last decade. It's not due to lack of effort — a new decentralized social network or digital identity platform launches, rebrands, pivots, and shuts down every month. And of course it's not due to lack of funding in the field, we can just safely state that.

What is stopping all those projects is a fundamental mismatch between what a typical blockchain offers and what application users learned to expect from their Web2 experiences.

It's not hard to name the reasons why: dApp UX can be best characterized as "death by a thousand papercuts." If your transaction takes seconds to get onchain — no real-time collaboration or games for you. If you need to spend non-negligible amounts of money for posting those transactions — no social. If your chain can't process even a thousand transactions per second without stalling[1] — how are you going to serve a million customers simultaneously?

[1] https://cointelegraph.com/news/inscriptions-evm-frenzy-clogs-up-blockchains

Nothing we ask here is too crazy; nothing requires breakthroughs in computer science or order of magnitude faster CPUs. Still, blockchains are stuck at “seconds per block” times, “hundreds of tps” throughput, node designs favoring datacenter deployments (resulting in the emergence of a ridiculously centralized layer of “RPC providers”), sticking to HTML+JavaScript as the only option for dApp builders, and an overall lack of care when it comes to supporting modern, responsive, native-first, real-time applications.

Can we achieve 0.1 sec block times, process 10× transactions per second, wrap that into a native-first SDK with *batteries included* — all without sacrificing decentralization or requiring top-tier server hardware to even run a node? If we would, the dApps experience would be drastically different.

And indeed we can. Meet the **Quantum Fusion Blockchain Network**.

1. Tunneling over the dApp gap

How are we going to do this? With the power of friendship and a bit of systems engineering, of course!

Blockchains are multi-tiered systems; they are pretty novel as well. Ever heard "premature optimization is the root of all evil in programming"[2]? It is absolutely, 1000% applicable to the current state of the art blockchain development: system levels of the stack fall apart, each optimized away from the other by engineers who don't focus on the bigger picture.

<P.1>

Yeah, you've made your networking stack incredibly modular, all while reducing the performance overhead of such modularity by 80% — great achievement that will keep your GitHub full of stars for years. But maybe the final product doesn't need that modularity? Maybe your dApp users will only ever connect to your blockchain from their web browsers, and supporting Tor as a transport may only add unwanted configuration options for the developers of those dApps to be anxious about?

We claim that a competent engineering team can achieve 10×, and maybe even 100× improvements over the status quo. They can do this by choosing the right building blocks, removing overheads and unnecessary abstraction layers, while keeping an obsessive, *laser-eyed* focus on the endgoal: the world's first dApp blockchain which doesn't suck.

In this paper, we describe our systematic review of a modern blockchain framework's stack layers, the most feasible optimizations, and how each change will bring us closer to our goal: the world's first blockchain ready for mainstream, mass-adopted decentralized applications.

[2] https://en.wikiquote.org/wiki/Donald_Knuth#Computer_Programming_as_an_Art_(1974)

2. Building on Substrate framework

Our optimization journey needs to start somewhere. And by "somewhere" we mean the best blockchain-building toolkit currently available, because to advance the state of the art, one needs to start from one.

Substrate is modern, modular, and written in Rust (perfect for our heavy in systems programming task). It powers hundreds of production projects[3] and is extremely hackable, exactly what we need for our ambitious plans.

And performance-wise (not taking the systems that sacrifice decentralization or failure resistance for inflated benchmarks in comparison — no name-calling, but *you know what you did*), Substrate also holds up spectacularly well: 1,500 TPS[4] (of actual balance transfers, with 3 second blocks) is not something many systems can deliver, even theoretically or in a carefully-controlled lab environments.

3. Improvements overview

Anyone can grab Substrate and get the performance noted above, out of the box. But they're nowhere near "10 blocks, 10,000 transactions per second" numbers we need to finally make blockchains useful for any major non-speculative usecases. Here's how we'll get there.

3.1. SPIN consensus protocol

Launching a new PoS blockchain from scratch is hard. 33% attacks are real, creating a chicken-and-egg problem: for PoS to be secure, tokens need value, but for them to have value, PoS needs to be secure.

So we're going to do a reasonable thing and buy some PoS security from Polkadot. It is a well-established network whose main goal is exactly this: providing its ฿100,000 worth of privacy to projects paying a modest ฿0.7[5] fee for this service.

[3] https://substrate.io/ecosystem/projects
[4] https://x.com/gavofyork/status/1270025498580656134
[5] https://parachains.info/auctions/polkadot

The actual security guarantees we're buying for this price are much more sophisticated than what, say, being an L2 on Ethereum can provide: Polkadot supports validating the full contents of our blocks (instead of just certifying that "yep, I've seen a block with this particular hash"), and Substrate allows us to leverage that verification without writing a single extra line of code!

But on a deeper level, we don't want to outsource our consensus mechanism to Polkadot and be its (or anyone's) L2. We also find Polkadot-provided time to finality (estimated to be 12…18 seconds, depending on implementation details) too long (we're aiming for 10 blocks per second, remember?).

So we'll build our own finality gadget, which will be run on our own validators, much faster and for cheaper (meaning weaker PoS, but better usability). And only the blocks validated by this gadget will be "promoted" to Polkadot. We call this algorithm SPIN (Short-term Parallel Incremental Network agreement).

<P.2>

And as a developer building a dApp on QFN, you'll be free to choose your finality guarantees: selling a house with an on-chain escrow definitely warrants an additional 12 seconds wait for extra security to kick in, while purchasing a coffee (or a character skin for your blockchain-powered MMO) can happen almost instantly, with just a 0.2…0.3 seconds delay — at par with the best Web2 payment systems.

3.2. Sequencing and execution parallelism

Getting a transaction into the block (thus securing its position in permanent history) and having it executed and reflected in the state update are two separate processes. For QFN, we propose decoupling these processes into workstreams happening in parallel, so we can speed up block production (effectively: just collation of semantically-correct transactions in some deterministic order) and transaction execution. Block B_N only commits to the Merkle root of the state from block B_{N-1}, allowing parallel execution of the state update function with already-committed transactions — or not committing to any changes at all. The latter covers the possibility of transactions in B_{N-1} being too heavy to be executed in time for our block production deadline.

While such failure indicates the runtime authors miscalibrating transaction weights, we want our blockchain to be self-healing — so the block production deadline increases exponentially for each subsequent block missing its state transition deadline, while block capacity for the new (not yet enacted) transactions shrinks in the same fashion, ensuring the efficient "catch up" of our parallelized state transition function after the transaction ordering one.

<P.3>

On top of everything else, this mechanism, together with extremely tight deadlines for block production, makes selfish MEV much harder to execute reliably. And if the node operators would achieve the required speedups and begin capturing it reliably, this should just be treated as an indication that block production deadlines can be reduced even further (say, to 0.05 seconds). Thanks to the Substrate flexibility, this can be done seamlessly, without stopping/hardforking the chain, just by tuning some parameters of the runtime.

3.3. PolkaVM

Substrate's forklessly upgradeable runtimes is an incredibly powerful framework to expose custom "precompile" functions to be executed as part of the transaction. A chain can launch with some basic set of built-in functions, and support for advanced ones (zero knowledge proofs verification, custom cryptography, Merkle proof verification of other blockchains for seamless bridging, etc) can be added later, as demand emerges — all running at close-to-native execution speeds.

Many projects still find this process too slow and permissioned. And dApp innovation requires them to express their state transition logic freely and on their own terms. Consequently, we need smart contract support in QFN.

PolkaVM, based on the RISC-V instruction set, is the most advanced smart contract execution layer currently available. It is both close to real computer hardware (ensuring the fastest launch and execution speeds compared to other engines like BPF, Wasm, or EVM[6]) and established well enough that major compilers and programming/security analysis environments already support it. With a modest degree of ingenuity, one

[6] https://github.com/koute/polkavm/blob/master/BENCHMARKS.md

might even be able to run a COBOL program as a QFN smart contract! *(Implementation is left to the reader, terms and conditions apply.)*

And for the projects migrating from the previous generation of smart contract environments, PolkaVM ships with an EVM compatibility layer — allowing projects to port their existing Solidity codebases without rewriting (and re-certifying!) them. Such emulation has its performance penalties, but QFN is fast enough that this, emulated, execution might outperform EVM on some other, well-recognized blockchain platforms.

3.4. Verifiable offchain workers

Some usecases (machine learning, complex cryptography like homomorphic encryption, even some games) mandate long, CPU and data access-heavy computations. It's not feasible to put such tasks in our 0.1 second blocks, regardless of our execution speed.

Here's where the latest developments in zero-knowledge proofs come into play. An off-chain worker can prove, leveraging certain fancy cryptography (similar to what is suggested in [Heiss et al., 2022][7] and [Domenech et al., 2024][8]) that the computation was performed as specified. Here's your answer, it's 42, thank you very much, anyone can verify no cheating occurred by doing a *simple cryptographic rain dance* with the execution proof provided alongside the result.

By allowing any blockchain user to act on either side of such market and exposing both the current blockchain state and some auxiliary, crypto-verified storage for large blobs, QFN would allow dApp developers to seamlessly mix fast, efficient, atomic on-chain transactions with explicit ordering with longer running, compute-intensive offchain operations without hard block-imposed deadline, using QFN chain as a payment and verification layer for the latter.

For example, this enables one to buy an NFT with usage rights for a 3D asset and launch its rendering into a scene — both powered by the same QFN blockchain, using the same APIs.

[7] https://ieeexplore.ieee.org/document/9881809
[8] https://ieeexplore.ieee.org/document/10634421

3.5. Accessing the Web2 from Web3: zkTLS

As much as we would like to teleport to the glorious future where all web is web3, it's not yet there, not even close. And we won't get there at all if the transition would be a bumpy ride.

This is why QFN would incorporate a built-in zkTLS mechanism: a gadget to commit Web2 data on chain, leveraging the cryptography used to encrypt website content for the browser anyway. That green padlock in your browser address bar means that the page you're looking at indeed came from your favorite social network and hasn't been tampered with while in transit. It's only logical that your dApp (using some fancy cryptographic tricks like the ones described in [zkPass whitepaper][9]) can do the same when putting the same data on chain.

3.6. Networking: nQUIC and WebRTC

Historically, most p2p networks relied on custom-made protocols for communication between nodes. These protocols were designed a decade or even two ago, when rolling out a custom, end-to-end encrypted protocol was not only about potential efficiency gains, but the only option to get all the necessary features, like multiplexing (transmitting several simultaneous logical data streams over a single, reused network connection).

Now we have http3/QUIC, which does everything needed — and it powers most global internet traffic, with top engineers ensuring everything works optimally on every level of the networking stack.

It would be nice if we could've just used that decade of performance improvements and internet infrastructure tuning to our advantage (and, as a nice byproduct, make our node communication protocol stand out less in other traffic, since many actors are trying to censor p2p applications out there, for various reasons).

And we can! Our protocol of choice is called nQUIC and is exactly the same as what your browser uses now when you're reading this page, just with a minor tweak: since we know in advance which node has which encryption key (that's how we identify peers in

[9] https://zkpass.gitbook.io/zkpass/overview/technical-whitepaper

distributed systems, instead of relying on centralized DNS and identity provision systems), we can discard the unnecessary complication (and centralization) of TLS key exchange protocols. We replace it with the straightforward, battle-tested, KISS protocol called NOISE[10]. That's what that 'n' in 'nQUIC' stands for — and the most experienced teams in networking and cryptographic design (like NCC Group and Cloudflare) worked on the protocol's specification[11].

Unfortunately, you can't run a custom network protocol from a browser due to very valid engineering reasons. Defaulting to regular https/QUIC would require each node serving in-browser dApps and light clients to have its own domain name and a valid https certificate to match. This approach is complex, compromises anonymity, and ties our blockchain to centralized web2 elements like certificate authorities and DNS providers. Instead, we're using WebRTC, a browser-friendly peer-to-peer protocol widely used for in-browser video calls and multiplayer games. Our engineering team believes the effort is worthwhile to make it work for standalone "full" nodes of QFN outside browsers, since WebRTC's built-in support for networking quirks and corner-cases, such as UDP hole punching[12], is hard to match.

3.7. Storage, sync and light clients

Any widely used blockchain suffers from `history bloat`: without special cryptoeconomic incentives like Existential Deposits[13] (which ruin UX), every transaction adds some (even if just a few bytes) data to the state, kept and replicated forever.

Given the rapid growth of server (and even desktop) hardware storage capacity, it might not be seen as a big problem. However, it only doesn't if you don't account for the fastest growing segment of Internet users: mobile devices. Not only have they limited (frequently non-upgradable) storage capacity, but the mobile internet access costs and speeds in most places make "downloading 1TB of blockchain data to participate in this shiny new cryptoeconomical activity" impossible.

[10] http://noiseprotocol.org
[11] https://eprint.iacr.org/2019/028.pdf
[12] https://webrtc.org/getting-started/peer-connections
[13] https://dablock.com/guides/learn-about-existential-deposit-keep-alive-transfers-on-polkadot

To target that "next one billion" Internet users (as well as people glued to their gadgets while sitting on a toilet — meaning *effectively everyone*), we need to make low bandwidth/low storage devices first class citizens in our networking designs.

`Light nodes` don't store or sync the full history of everything. They just store enough to know how to tell truth from lies about the on-chain data, using the clever mechanism called Merkle proofs[14]. They need to download block headers, which is ~100 bytes per block — and this can be further reduced by an order of magnitude (using crypto-verified skip-lists) for cases that can tolerate a few seconds of latency.

Historically, light clients have been an afterthought in most blockchain designs; extracting and providing a Merkle proof from blockchain storage takes a non-negligible amount of resources. Without a carefully designed set of storage optimizations and light client protocol improvements, it's only expected that light clients in those networks are heavily throttled, limited, and generally perform way below their theoretical capabilities.

In QFN, we aim to support millions of users interacting with dApps simultaneously. If they are not running their own light nodes, they would be at the mercy of the few centralized RPC providers (as seen in some other major networks), which introduces both privacy[15] and censorship[16] risks.

Instead of the typical limit of "100 light client connections per full node," we aim to raise the bar at least 100×. We will do this by tuning the network protocol used by light clients and leveraging our team's expertise in RDBMS storage engines to make the overhead of extracting and packaging a Merkle-proven answer a lighter task any full node shouldn't fret too much about.

Designing DB storage engines for real-life applications is more of a dark art than science. So, we're taking a benchmark-heavy approach for each and every minor design decision here and will be (in our team's usual rigorously-transparent fashion) constantly publishing articles and writeups with our latest findings and relevant codebase improvements.

---

[14] https://decrypt.co/resources/merkle-trees-guide-explainer-blockchain

[15] https://www.digilol.net/blog/chainanalysis-malicious-xmr.html

[16] https://cryptoslate.com/metamask-blocks-ethereum-transactions-in-several-jurisdictions-citing-compliance-issues

3.8. RPC layer: Cap'n Proto

As much as we'd like to have light clients as the dominant mode of connecting end-users to the QFN blockchain, in some usecases even "light" is not light enough. There are probably better ways for a microwave oven or fitness bracelet to interact with on-chain data than having the device sync with the network state continuously.

The key points about such embedded usecases are that it's a) quite sensitive to the chosen protocol's performance and b) not constrained by modern web browser technologies. So there's no reason to pick the traditionally-used JSON-RPC (and suffer its inefficiencies and quirks, like having to pass blobs of data as hex-encoded strings instead of much more compact "raw" bytes). We chose the state-of-the-art RPC protocol (in terms of performance and encoding/decoding efficiency), which is Cap'n Proto[17].

Not only does it feature a zero-copy decoding (meaning the data sent "over the wire" doesn't need to be transformed for future consumption — what you get from the network is exactly what you need), but also supports `promise pipelining` [https://capnproto.org/rpc.html], allowing our microwave to send its first query ("who is the address of on-chain wallet named 'Bob'?") and immediately follow up with the second one ("and by the way, what's the balance of that wallet?"), without waiting for the first answer (the "that wallet" part is what programmers for some reason call "a promise" in their code*; Bob is a really trustworthy guy who can be trusted on his promises, it seems. Maybe he's an uncle of yours?*).

<P.4>

But, you may ask, how does the microwave know which server to ask? If it's just the good old centralized DNS query, we end up with the classical centralized "client-server" architecture — just with extra steps. Can we do better?

We definitely do. Of course, QFN blockchain might get some centrally-run RPC infrastructure: for example, your microwave's manufacturer could set one up for their devices to use as a fallback. But we want our *microwave-development SDK* to first try and discover local QFN nodes (like that QFN-enabled messaging app you keep running in

[17] https://capnproto.org

the system tray on your laptop), and only reach out to global, centralized ones as a fallback.

There's a whole set of technologies aimed at local services discovery, ranging from using "basic" UDP broadcast packets to more sophisticated tech your router might or might not support, like mDNS[18] and UPnP[19]. Picking the correct protocols to support in our SDK cannot be done "off the bat" and will be a long project of trial, error and feedback from early beta testers — but rest assured this is not a feature we're willing to compromise on.

3.9. Internal optimizations

And of course, our design improvements over the "baseline" Substrate don't end here. The current framework has already accumulated multiple decisions and tradeoffs we consider to be "legacy": there's nothing inherently wrong with them, but they're kept in the codebase mostly for backwards compatibility reasons — and we, developing QFN from scratch, can name several dozens of places where we (with our specific usecases and design goals in mind) can code something more efficient: more optimal "hot path" code, less overhead via hardcoding a certain design decision (like picking BLAKE3[20] as the only universal hashing primitive across the codebase), easier to debug, and so on.

None of these choices (well, except the BLAKE3 one; *that shit is lit*) would bring a spectacular speedup on its own — but those are expected to accumulate, resulting in a solid 2×..3× improvement over the generic Substrate case presented in the previous chapter.

4. Things we keep from Substrate

From the chapter above, readers might get the impression that we leave no stones unturned, aiming for a full internal rewrite of the Substrate framework, leaving it in our codebase by name only. Would it have been better to just write our own blockchain beginning with an empty Makefile?

[18] https://en.wikipedia.org/wiki/Multicast_DNS
[19] https://en.wikipedia.org/wiki/Universal_Plug_and_Play
[20] https://www.ietf.org/archive/id/draft-aumasson-blake3-00.html

Many teams take this approach (after all, programmers love building their own lunaparks from scratch). However, this wouldn't be wise, since Substrate blockchain building framework is an extremely well-thought-out set of foundational libraries, and we will reuse their state-of-the-art features and battle-tested codebase. In this chapter we highlight the most important things we are taking for granted with this approach.

4.1. Forkless runtime upgrades

When people criticize "code is law" as a maxim, most frequently they imply "*immutable* code is law." And they are correct in that. Even if the original system was flawlessly engineered and implemented without any bugs, continuously maintaining relevance against something as fast-changing as worldwide socioeconomic structures is not something that anyone would manage to hit in a "once and forever" fashion.

Our system needs to evolve alongside its surroundings; having a built-in way to extend, improve, or fix bugs in our blockchain's core without dealing with contentious forks and the status quo bias of huge capital allocation on chain development is such a natural requirement it's strange that no other systems developed anything comparable to Substrate's forkless runtime upgrades mechanism.

The idea is simple: let's encode the canonical rules of our blockchain not in some "colored paper" PDF, but in an executable binary (expressed in Wasm, for platform portability and universality), and store that blob on the very chain it describes. Our protocol should also cover "meta-rules": 1. An algorithm for proposing and accepting new rules through democratic consensus among blockchain stakeholders. 2. A coordination mechanism allowing every node of our blockchain to switch to this new `Runtime`. 3. A clearly defined opt-out process to allow participants escape the potential hostile power capture: eg, you'd need to restart your node with a specific CLI parameter to skip one particular runtime upgrade (and thus create a fork of the network).

This way, politically engaged stakeholders will use well-defined on-chain mechanisms to govern the evolution of the Substrate-based chain, while inert node operators and institutions would effortlessly stick with the outcomes of that governance, without worrying about future changes affecting their operations or whether to oppose them to improve their operational bottom line.

One wonders why we don't see this everywhere — well, at least we will see this in the QFN blockchain.

4.2. Ecosystem of pallets (including ever-useful Assets and Multisig/Proxy)

There are so many behaviors in the blockchain world that aren't novel or shiny. Think of balances and token transfers: they need to be there, they need to be extremely fast and reliable, with a codebase reviewed by top security professionals — and that's it; there's nothing inherently controversial in the design of such a system, or any opportunity for algorithmic optimizations (those are mostly basic arithmetics performed on balances) involved. There is also a huge benefit in having this as standardized as possible, so exchanges can reuse their integration code from some other similarly-built coin, and developing and certifying a new Ledger app from scratch to sign such transactions won't be necessary.

Luckily, Substrate got us covered with FRAME[21] — a collection of ready-for-production implementations of the most common blockchain primitives. These range from basic assets/balances to the support of atomic swaps, chain-native account abstraction primitives (with built-in multisignature wallets and proxied accounts). They are free for us to pick, reuse, and improve as we see fit.

An absolute must if you don't want to waste your world-class blockchain engineers' time reimplementing basic bookkeeping the 1000th time.

4.3. SCALE and metadata (and existing hardware/software wallets)

Input deserialization is such a major source of software bugs and vulnerabilities that it made it into its own category in the OWASP Top 10 in 2017[22]. And blockchains fundamentally a) interact with a lot of unfiltered, untrusted input from arbitrary third parties (anyone can send a transaction, that's the whole point), and b) pay a huge premium for each extra byte required by the transaction serialization format, since each such byte would be kept for decades, replicated over thousands and thousands of nodes.

[21] https://docs.substrate.io/reference/frame-pallets
[22] https://owasp.org/www-project-top-ten/2017/A8_2017-Insecure_Deserialization

On top of this, codec design is a slowly-moving field (unlike, say, algorithms powering decentralized consensus engines), and reusing some existing well-established encoding protocols is not going to set us back against competition.

So we can do better than rolling our own codec and requiring our own tooling for every programming environment in the world. The Substrate codec, SCALE (Simple Concatenated Aggregate Little-Endian)[23], is as good as it gets. It is pretty compact, designed specifically for blockchain usage by the same people who previously designed RLP, which is still used in every Ethereum-compatible network out there. By now SCALE has been deployed to multiple multi-billion dollar blockchains for years, meaning it has been tested and fuzzed exhaustively, with no stones unturned.

<P5>

It's not enough to know *how* to encode things. Communicating parties need to agree on *what* to encode. This isn't a problem when two identical versions of the same blockchain node talk to each other. But what about a no longer maintained web frontend for a blockchain that underwent significant runtime upgrades? Section 4.1 highlights why QFN having runtime upgrades is a desired feature, not a bug.

Substrate has us covered here as well: not only every runtime can expose formally-specified `metadata` (a form of dictionary, detailing how to parse each possible blockchain-specific message for this particular runtime version) — but the runtime itself is metadata-aware, meaning a hacker can't mislead users into believing the transaction encoded as `` `0×abcd` `` sends 1 coin to Bob while actually sending all of them to Mallory. The runtime will reject transactions built on false metadata assumptions, rendering the whole class of attacks impossible. Who doesn't like the whole classes of attacks being rendered impossible? Hackers, not us in the QFN team.

4.4. Ecosystem tools: block explorers, dev tools, Metamask snap etc

Ok, so we have a pretty standard network protocol carrying over data serialized in a well-known codec, to be interpreted together with some machine-readable metadata

[23] https://github.com/paritytech/parity-scale-codec

that is already supported by existing blockchain-interaction libraries for most frequently used programming environments. What does it get us?

Well, *everything*. Block explorer? There are several opensource codebases to chose from (Subscan[24], Polkascan[25]). Interactive developer tools for dApp builders to experiment with before writing production code? We are well covered (PolkadotJS[26]). Helm charts for node deployments in Kubernetes, together with all necessary telemetry and monitoring, and even optional support for testnet management and a faucet? All there (paritytech/helm-charts[27]). A `sidecar` to be used by exchanges to effortlessly integrate your token into their existing systems? Aye (paritytech/substrate-api-sidecar[28]). There's even a Metamask "snap"[29], for your users to pretend they're using their grandfather's old boring Ethereum while actually living at the bleeding edge.

Substrate: come for an ultra-performant blockchain framework, and stay for the full range of turnkey solutions for bootstrapping the whole ecosystem surrounding it!

5. SDK

So we've built our blockchain, and it turned out to be even faster than we anticipated. The problem is — it's useless without other developers building their dApps on top of it. For that, we need an SDK (Software Developer Kit, or, how some members of our team call it, *Several Dozen Koffees*).

What properties should the SDK have to ensure that the apps built on top of it are faster, snappier, and more secure than today's average blockchain-powered dApp? *(Not a high bar to clear, we know.)* How do we even approach this question?

[24] https://github.com/subscan-explorer
[25] https://github.com/polkascan
[26] https://polkadot.js.org
[27] https://github.com/paritytech/helm-charts
[28] https://github.com/paritytech/substrate-api-sidecar
[29] https://github.com/ChainSafe/metamask-snap-polkadot

5.1. Light-client first

As discussed above, we don't want our ecosystem to become a hostage of a centralized RPC provider providing the only way for the majority of the dApp users to interact with our chain.

This means no matter which tech stack (browser, mobile, desktop, etc.) our SDK targets, the codebase of our light client should be trivial to compile and embed in all of them. Enter Smoldot[30].

Rust is a pretty universal language. You can compile it to any of the supported target platforms[31], from server-grade 64-bit CPUs to exotic 16-bit microcontrollers, GPU-specific microcode and of course platform-independent Wasm bytecode, if you follow certain development guidelines. At the moment of writing, it's not yet clear what it means to have a blockchain light node compiled for the PTX bytecode to be executed on NVIDIA GPUs. But the possibility is there. Maybe Apple VisionOS will become a relevant platform for some blockchain-powered metaverse project in the future?

No matter which platform you want your SDK for, betting on Smoldot for light client operation is a safe, reliable bet. And *smol* in Smoldot hints at low system requirements; our internal tests indicate the same, so no worries that your users can only use your fancy QFN-enabled dApp on the latest, largest, most expensive Pro model of the smartphone — any reasonably modern one will do just fine, as well as any web browser.

5.2. Batteries included

Ok, cool, your dApp can read and write from the blockchain without compromising its users' security and privacy while doing so. Now what?

Well, taking all other building blocks the SDK will provide — and building a wonderful castle (or whatever your heart desires). We aim at providing a "batteries included" experience — once again an area where Substrate ecosystem supplements our efforts with enough pre-existing solutions: encryption and key handling libraries, common hardware wallets support, high-level API for QFN-provided features (like Verifiable

[30] https://github.com/smol-dot/smoldot
[31] https://doc.rust-lang.org/rustc/platform-support.html

Offchain Workers), useful app-building primitives (like the fastest CRDT[32] implementation we will be able to ship) and even a special scripting language (also known as "DSL") for expressing privacy-preserving custom logic for your app, to be converted into zero-knowledge execution on-chain.

While the chain itself can handle offloading your "big data" computations to the QFN-provided offchain workers, storing that data on-chain might be a bad idea: it's called *big* for a reason, and when storing data on-chain, every byte counts. That's why the QFN SDK will ship with iroh[33] bindings for integration with the most advanced content-addressable distributed file-system currently available, so you have a way to store and distribute your Big Data Files™ in a crypto-verifiable way, no matter how big they actually are — all transparently accessible from your QFN-powered apps.

5.3. Web now, native later

The consensus within our engineering team on this topic is unusually strong: web browsers are one of the worst application development platforms in existence. They are slow, laggy, and limited in ways they interact with both user inputs and the computer's hardware… They are also the most widespread ones, and the reason why there's a word web in "web3".

Therefore, our engineering priorities are straightforward: the first platform to be supported in the QFN SDK should be a modern, HTML5-enabled web browser. Yeah, we'll have to tame Smoldot to behave under the memory and performance constraints of the JavaScript engine; ensure WebRTC connections to other blockchain nodes work via mobile internet; and expose iroh and cryptography bindings to browser-based JavaScript. It's good to be standing on the shoulders of giants here, and not having to fight all those fights alone, building codebase from scratch.

That said, we aim bigger and bolder than being just another JS-based toolkit for blending web2 and web3 together. So far, there have been no significant attempts at launching a native-first application platform for blockchain-powered dApps on mobile devices. What can it look like? Roblox, but for adults? Google Docs, but for DeFi trading strategies? TikTok for AI-assisted freelancers? Let's discuss potential usecases.

---

[32] https://crdt.tech

[33] https://www.iroh.computer/docs/components/blobs

6. Areas of potential application

Currently, blockchain-powered games are essentially games in name only. There are no major productivity apps that truly leverage the distributed, censorship-resistant power of blockchain networks. No reliable way to hire a freelance graphic designer, pay them, and track their reputation using a decentralized ledger. No major distributed social network or chat app. Even online auctions and classifieds aren't replaced with p2p-provided crypto verified matchmaking and escrow services. In short — web3 is still in its infancy.

In our opinion, the current web3's technical limitations just make them not competitive enough with all the bells and whistles consumers have learned to expect from their typical web2 counterparts.

What if the users didn't have to wait a dozen seconds for their changes to reflect in the blockchain-based collaborative document? What if there was a chain that, by design, avoided a single point of failure (and in effect, censorship) while allowing citizens of any, even the most repressive, country access to a global network of social and professional connections? What if there was a blockchain-powered job market with a search results page that didn't feel laggy when displaying a few hundred matching CVs/job offers?

This is exactly what we're aiming at with QFN: 100msec transaction latency; true scalability, adopting the latest web2 technologies for our usecases; building a network that would sustain a million light nodes participating; native-first SDK with batteries included, allowing global developers to roll out mobile apps with the same ease they are now doing with web-based ones.

<P6>

What would you build on top of Quantum Fusion Network?

Appendix

How Blockchains Work

Blockchains are state transition functions, executing transactions over a replicated state stored in distributed ledger. (Bear with me, all this will come useful a bit later.)

Ledger is run by peers, which communicate over global networks (most often — Internet), and those peers expose their state for the consumer dApps to read from and interact with. It's that global shared state (and mutations of it, happening in concert and fast enough) that is actually important in the blockchain building, and transactions/blocks/peers are just the state of the art technical framework to achieve this.

Our aim is to build a ledger, which would be good enough for the broadest level of real-world applications, without introducing any `failure modes` which will make this unreliable.

Blocks in the blockchain is a way to order the global execution of transactions — without global order things like `double spends` are impossible to avoid.
We only know how to have a single block producer every time, and it needs to be circulated (ideally: unpredictably and permissionlessly). No circulation (or no way for a random entity to enter the pool of the block producers) — no `censorship` resistance (both on technical, by hackers, and organisational, by hostile economic actors, levels)

Finality is needed for any real-world economic activity: we need to know when transaction is "done", and block producers can create forks in history even when chain is not being attacked. With no finality, we can never be certain that our transaction won't get overwritten by a `reorg`.

Consensus provides both block authoring "lottery" and finality mechanisms: but not necessarily by a single algorithm. Consensus failures (like `chain splits` or `finalization delays`) are common both for bad computer science and engineering mistakes.

Lifecycle of the transaction:

- Transaction goes into the mempool, gossiped across block producers
- Transaction gets its spot in the block
- Transaction gets executed, and the result (success/failure/outofgas) is recorded in the state
- Block is getting transmitted across other nodes, which update their state correspondingly
- The state is confirmed by the finality gadget of the consensus.

There's no requirement for blocks to include the updated state immediately (even if it's useful in high-blocktime systems), and multiple finality gadgets can coexist (provided they are nested, and not parallelized — to avoid potential chain deadlock).