The technical challenges of building a decentralized application

Tips for building your dapp from the creators of CryptoKitties

Howard Tam
Sep 23, 2019 · 10 min read ★

Building CryptoKitties was a lot like herding cats.

Sure, there are best practices for building a decentralized application (dapp), like keeping them simple and secure, but there are a lot of challenges and unknowns too.

- Frontend engineers and user experience designers need to communicate state transitions to users, because *god help us if they don't*.
- Do users need to know what a gas limit is? Maybe!

- Then, there's the transactional nature of each interaction, with Ethereum emphasizing the passage of time to make asynchronous processing a necessary aspect of any decentralized application.

In this article, we explore how the technical architecture of CryptoKitties evolved, highlighting the learnings that we have made since its inception.

Interacting with Ethereum

To start with, let's briefly visit some considerations developers should have when developing a dapp.

Each interaction with the Ethereum blockchain that involves state mutations requires a transaction. From the perspective of the application, all it tries to achieve is simply:

1. send a transaction specifying details of state mutation
2. perform actions depending on the success / failure of the transaction

How does an application send a transaction to the blockchain? Well, it needs to be able to talk directly to an Ethereum node so that the proposed state change to the blockchain can be propagated throughout the network. One tool to facilitate this is web3.js, a collection of libraries for JavaScript developers to make JSON-RPC calls over HTTP, WebSocket or IPC toward any Ethereum node that exposes an RPC API.

There are various ways an application can determine updates to the blockchain states. One simple approach might be to poll the state by periodically making JSON-RPC calls to the eth_call method, which involves directly sacrificing application UX (degraded frontend performance) to achieve engineering simplicity.

A different approach is to listen for state changes via a subscription model. Specifically, events are "emitted" to notify clients of the blockchain's updated state — they are essentially logs made available via each transaction receipt. The logic within a smart contract dictates where these logs are produced and are commonly used to communicate state changes.

In practice, applications can take advantage of web3's recent release which exposes event emitters for WebSocket connections. For example:

- web3.eth.subscribe function allows you to subscribe to incoming logs, pending transactions, and
- web3.eth.sendTransaction and contract methods return “PromiEvents” — these are normal promises with the ability to pass in callbacks to its on, once and off functions

Architecting CryptoKitties

We needed more than just a frontend

While technically it is possible to implement only a frontend application that handles everything from game logic and UI to sending an Ethereum transaction and listening to the emitted events, it can be impractical. Regularly giving birth to gen-0 CryptoKitties requires periodically calling the smart contract — the frontend should not have to care about this. Additionally, there exists a lot of game logic that a server would be better suited to handle instead of smart contracts — eg. rendering kitties according to their genetic makeup.

Given the inherent asynchronous nature of blockchain interactions, there were many benefits to building a processing-heavy CryptoKitties backend that served resources to a rather lightweight application frontend, thereby delivering a faster, smoother *perceived* experience. One example uses a worker to listen for Approve events before submitting transactions to fulfill a bidder’s offer on a CryptoKitty (the owner needs to first approve the Offers contract to transfer the asset). Another uses a worker to respond to AuctionSuccessful events in order to keep the leaderboards updated. On the CryptoKitties frontend, transaction objects are sent to Ethereum nodes via web3 and the retrieval of both off-chain and on-chain data is done via a RESTful interface with the backend API server.

Takeaway: when complicated functions become a backend’s responsibility, the frontend can focus on delivering the best user interface and experience.

Pay attention to gas

Gas limit and price specifications for each Ethereum transaction can drastically dictate the success of an attempt to update the network state. On one hand, gas limit needs to be sufficiently high in order for the smart contract logic to be executed (more gas is required to execute more complicated logic) — if not, none of the intended state changes will occur at all. Then, gas price directly translates to the size of the reward

given to miners should they process the particular transaction (higher gas price means greater reward per unit of gas). As miners decide on which transactions to process, a transaction with a rather high gas price will be quicker than average to be processed — but if set sufficiently low, the transaction may be left in a node's mempool indefinitely (a "stuck" transaction) as miners end up never processing them. For these reasons, running a worker is essential for monitoring any pending transactions that exist in the mempool and re-sending any "stuck" transactions at higher gas prices.

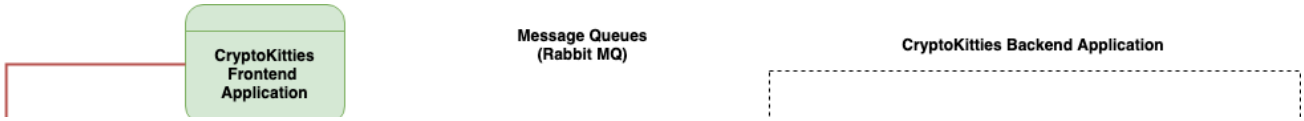
Takeaway: gas specifications impact the success of your transactions and workers will be necessary to supervise transactions until they are mined.

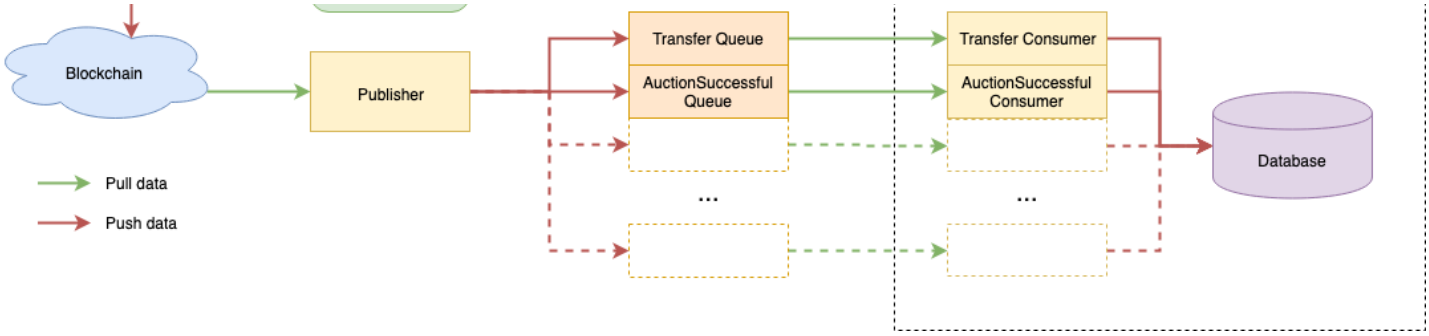
Subscribe to state changes

Multiple workers are also utilized to act upon state changes on the blockchain. One worker listens to the events emitted from the CryptoKitties smart contracts and proceeds to publish these messages to a cluster of Rabbit MQ instances. Another subscriber would be deployed to consume messages from the Rabbit MQ queues, which were arranged by event name. By utilizing a message queue, the contract events were able to be routed such that each event would be processed only once.

The database schema was designed to collect as much information as possible from the blockchain. On top of processing event arguments, the event's associated contract address, block number, block creation time, transaction hash and state were persisted to our PostgreSQL database. Indeed, by acting like an audit log it has provided great efficiency boons to our debugging process. A major motivation also stems from the limited functionality of the blockchain nodes' API — it cannot perform any complicated queries (eg. table joins) as relational databases naturally provide. To work around this, a more effective approach was to replicate as much relevant blockchain states onto traditional, familiar tooling such as PostgreSQL (after all, storage is becoming much cheaper), which resulted in significantly lower barriers for new developers to contribute as well. By creating this abstraction, the application business logic benefits from having a single data source, through which off-chain and on-chain interactions funnel.

Takeaway: running workers with a message queue like Rabbit MQ is one effective way to subscribe to blockchain state changes.

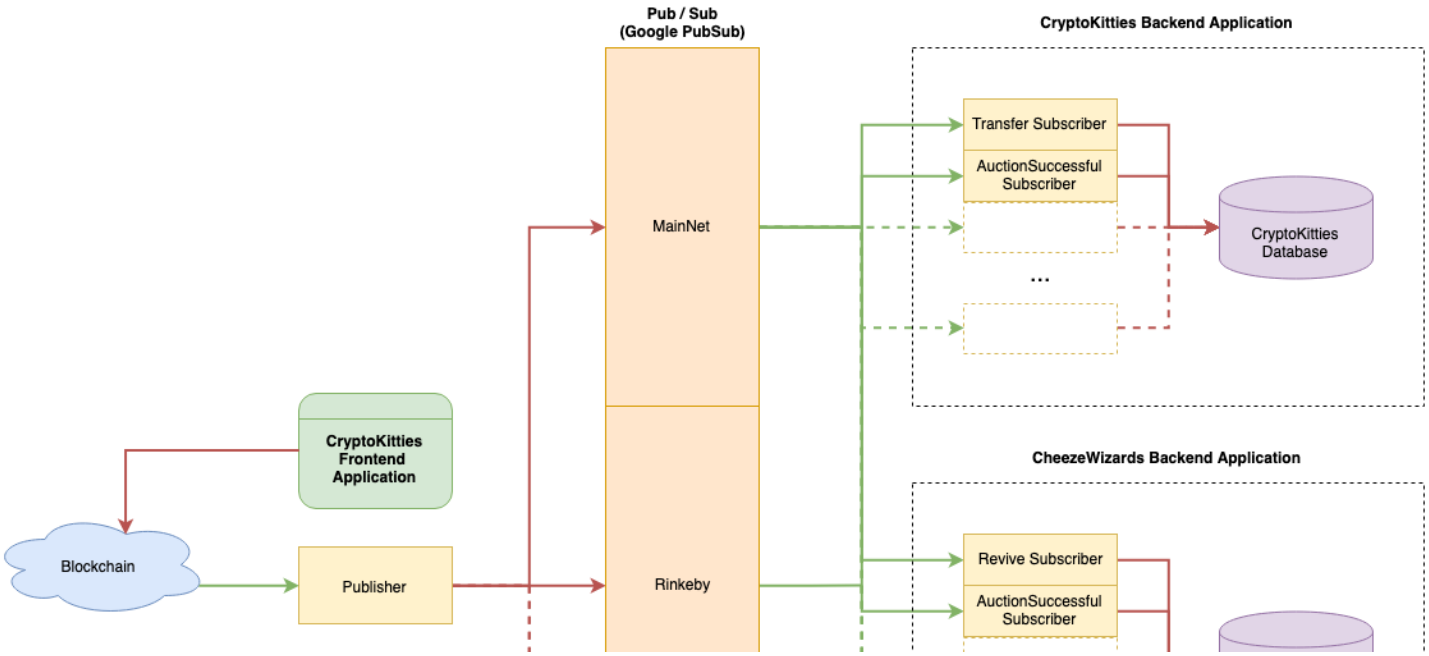


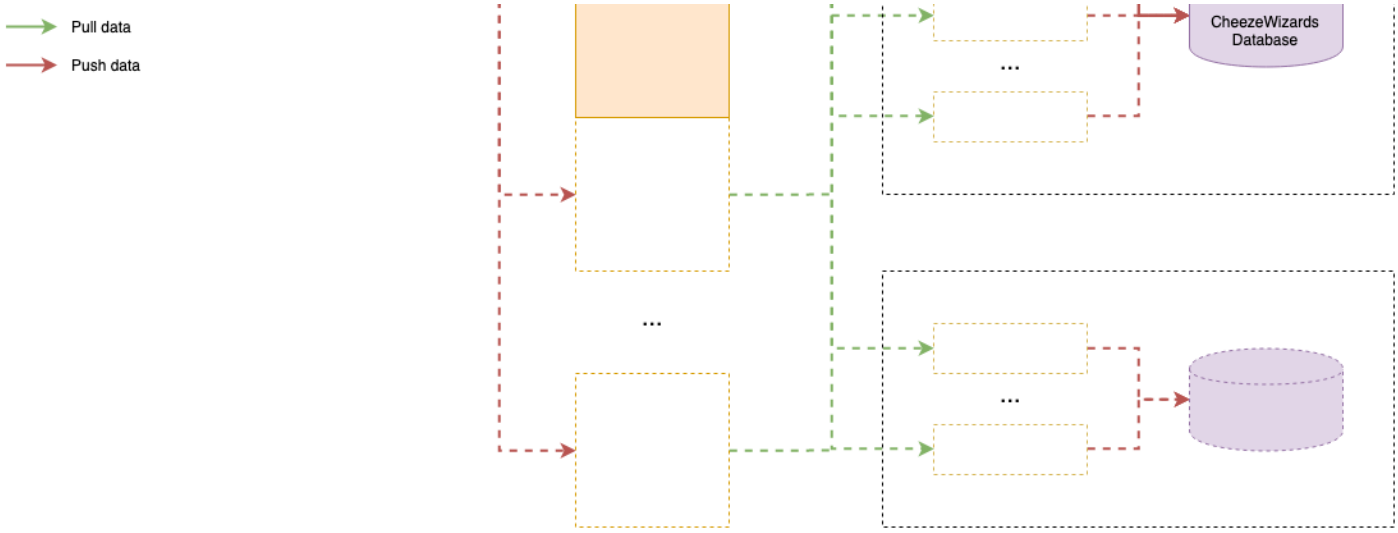


Architecture of Rabbit MQ Integration

Use PubSub to scale horizontally

Over time, the number of Dapper Labs projects grew, and we needed a scalable solution for processing blockchain events. Instead of each project using their own message queue cluster and workers, a PubSub service hosted by Google Cloud was chosen to satisfy our fan-out messaging requirements — its low latency meant little overhead as we created this layer of abstracted subscriptions above the layer of blockchain nodes, and its persistence of event ingestion enabled transactions to be replayed whenever necessary. Contrary to how the Rabbit MQ publishers functioned — filter for events from particular smart contracts — the PubSub publishers read the transaction receipts from every new block that got mined. As the publishers no longer assumed knowledge regarding various business functions, it became incredibly flexible for cross-domain applications to plug into various streams — applications can simply subscribe to any relevant topics for filtering contract events that mattered. Each subscription group is made responsible for transforming the messages according to its specific domain, without fear of affecting the message schema for other subscription groups.





Architecture of Google PubSub Integration

Google PubSub is a service that delivers each message at-least-once, without guarantees for ordering. As such, we have had to design our subscription groups to be idempotent when handling duplicated messages. One way we achieved that was the use of transaction hashes as a primary key for some of our tables. Though this was a fast solution, duplicated messages manifested as increased errors throughout the application and resulted in more logic to handle them.

At the same time, various methods in the data access layer were made to upsert instead of insert, which provided a more graceful approach to handle duplicated, and even missed messages (events that were incorrectly processed). Instead of replaying missed PubSub messages, upsert methods can gradually bring the database to an eventually consistent state. Of course, this relies on the promise of future activity that overwrites similar sets of data. For example, imagine an auctions table managing a single record of auctions per kitty:

- For an active auction, say the AuctionSuccessful contract event was missed, so the kitty has been handed over to the new owner address in a kitties table, but the sale auction is still attached to it.
- The database can become consistent once there is a new auction being made on that kitty, granted the AuctionCreated event is captured — this is because the new auction data would overwrite data in the existing record for that kitty.

Takeaway: adopting a PubSub scheme facilitates horizontal scaling as business logic is now within subscribers' scope.

Nuances of order handling

The order in which messages are processed matters significantly — 2 Transfer events arriving out-of-order can cause different interpretations of asset ownership in the off-chain system. Initially, our approach towards handling out-of-order messages was, rather, not to do so. The subscribers would stop processing, signal errors regarding inconsistencies between the event and the database, and occasionally intervention would be required to manually cherry-pick various messages from the queue. Consider the following scenario:

1. User A gifts a cat to User B, emitting a Transfer event, then
2. User B gifts the same cat to User C, also emitting a Transfer event

If the subscriber processes 2. first, then kitty ownership in the database should change from User B to User C, yet User A is still marked as the existing owner (since 1. has not been processed). Once processing is halted, it usually takes several steps to even decide if messages had arrived out-of-order or were simply missed — we would manually trace the Etherscan transactions that affect on-chain kitty ownership, then proceed to check logs upstream in the message processing pipeline, before executing the corrective action. These incidents were especially inhibiting to the player experience. Fortunately, the CryptoKitties frontend verifies state by reading the blockchain before sending transactions, which prevents users from wasting gas from making misguided decisions while navigating through inconsistent data displayed in the game.

With the Offers feature (released Dec 2018), a different approach was considered which resulted in a slightly better experience. Since a CryptoKitties Offer can undergo many state changes, there were many scenarios that missed or out-of-order events can easily disrupt the application. To name a few,

- an Offer can be created, then outbid (createOffer)
- an Offer can be created, then updated (updateOffer)
- an Offer can be created, then fulfilled (fulfillOffer)
- an Offer can be created, then cancelled (cancelOffer)
- an Offer can be created, expire, then refunded (batchRemoveExpired)

By programming rules for various state transitions in the backend system, the application is able to fill in the intermediate, “missed” states whenever necessary.

Processing an OfferFulfilled event alone meant that either an OfferCreated event was missed or has yet to arrive — the application can then backfill into the table of created offers. Though this approach generally resulted in having to handle more edge cases, increased code complexity, and ultimately still wasn't a catch-all solution — for example, given an OfferCreated event and subsequently an OfferFulfilled event at a higher offerPrice, there would be no way to tell if there were any outbids on the original Offer created; there was a lot of incomplete information.

Provided that there are no incidents of chain reorganizations — this risk is minimized as blocks are only read when they have a sufficiently high number of confirmations — messages arrive out-of-order at subscribers mainly because of Google PubSub. Their documentation suggests making use of sequence identifiers as part of a synchronous step, which may involve incorporating block number for example (though that alone is insufficient as multiple transactions can exist in the same block). Indeed, low-latency, scalable publishers that guarantee total ordering in coherence with the network state changes is quite desirable. By having in-order messaging downstream from the publishers, the rest of the backend system benefits by becoming more lightweight — it need not understand state transition rules that smart contracts enforce, nor have to deal with message ordering. It can "blindly" accept whatever comes from the upstream pipeline.

Takeaway: pay attention to how the lack of ordering guarantees from Google PubSub might affect your application.

What's next?

To summarize, here are some of the main topics that were discussed throughout the article:

- We briefly touched on the various ways applications can interact with the Ethereum blockchain
- Then, we evaluated some performance shortcomings of a frontend-only dapp and noted how complicated game functions might be better handled by a processing-heavy backend
- Then, we described how backend workers are necessary to guarantee transaction delivery and to act upon blockchain state changes, noting the utility of replicating transaction information onto our PostgreSQL tables in enabling complex queries

- Then, we described the transition towards a PubSub pipeline as the organization needs became more diversified, which involved moving domain-specific knowledge further downstream of the processing pipeline, as a subscriber's responsibility (instead of publishers to the message bus)
- Then, we described particular nuances of developing with Google PubSub, specifically the challenges of ingesting blockchain-related messages whilst handling duplication and no order guarantees

Keep a lookout for more posts in the future! There are a few other notable architectural changes that warrant their own technical deep dives.

. . .

Have you built a dapp too? We'd love to hear about your architecture!

A lot of our challenges building CryptoKitties informed our approach to Flow, a developer-friendly blockchain Dapper Labs is helping to create. You can learn more about Flow here!

. . .

Acknowledgements

Many thanks to Nick Salloum, Jordan Schalm, Fabio Kenji, Stacey Lin, Leo Zhang, Eric Lin, Kan Zhang, Jordan Castro, Alan Carr, Ian Pun, Daniel Anatolie and Bryce Bladon for reviewing this article, suggesting improvements prior to publication.

Blockchain Cryptokitties Software Architecture Ethereum Dapps