Bacalhau research deliverables

Luke Marsden, in collaboration with @dignifiedquire, Alex Terrazas, 2022-03-24

# Payment, incentives & rewards

Bacalhau nodes will typically run alongside IPFS and/or Filecoin nodes.

Determining the exact method of payment is out of scope of this document.

We have evaluated TrueBit (ref this report) but it does not seem to actually be in use in Golem. The fact that TrueBit requires a computation to be subdivisible into arbitrarily small pieces makes it impractical for use in Bacalhau. What's more, we can't see that Golem actually implements any verification currently, either for its legacy WASM runtime or its newer Docker-like VM runtime. We will continue to research this. If anyone has any more information on this please let me know (@lukemarsden on Filecoin Slack, #bacalhau channel)!

Payment, incentives and rewards will be mediated by a smart contract (prototyped in Solidity and maybe ported to Rust) running on the FVM mainnet when FVM launches.

The smart contract interface is defined in https://github.com/filecoin-project/bacalhau/blob/main/pkg/transport/types.go#L12-L13

Bacalhau compute nodes and requester nodes will coordinate by subscribing to the smart contract.

Hopefully FVM will be efficient enough not to need a layer 2 e.g. zk rollup system on top. However we will need to be careful since paying for compute over time relies on many tiny payments. The gas fees need to not exceed the fees actually paid for compute.

Since Bacalhau is not a LURK based system (at least initially; however it hopes to provide a framework for such systems), Bacalhau will target a "spectrum of trust" that builds on top of existing human trust systems.

This will provide a base level of trust for participants.

We will also do programmatic verification. Rather than TrueBit's approach which requires a separate verifier role, we will have clients drive the verification of results. This might not make Bacalhau results globally verifiable, but they are verifiable to the people who are paying for them, which is sufficient. Having clients verify results (by requesting some of them are re-done) simplifies the system since they are already incentivised to get correct answers, and so no "forced errors" are required.

Verification protocol

To be implemented as a smart contract.

Not "optimistic" exactly (no separate verifier role) but sort-of. The clients randomly choose to verify work by getting it re-done by random participants in the network. The rewards for verification jobs are high enough that everyone should want to do them. Clients and servers both stake money. The punishment for clients of not correctly verifying hashes is that they get slashed. The punishment for servers of not correctly doing work is that they get slashed.

Below read "client" as shorthand for "trusted (by client) requestor node on behalf of client".

- Compute nodes (servers) stake a significant amount of money which they lose a fixed amount of if they fail verification of jobs. They must keep their staked amounts topped up to continue to participate in the network. This minimum will make Sybil attacks very expensive for servers.
- Client requests N pieces of work to be done with concurrency = 1 (either all at once, or gradually as they request work).
- Clients escrow (stakes) payment for the job into the smart contract from their wallet balance, which must not drop below a minimum. This minimum will make Sybil attacks very expensive for clients.
- Payment is per CPU-hour and mem-GB-hour? But then we need to solve attacks around compute nodes throttling jobs.
    - WAS: Payment is for the entire job, not per time. This helps reduce the total number of transactions that need to pass through the network, and reduce the impact of gas fees. The client should reasonably be able to estimate the amount of time the job will take, and can specify an upper bound so that compute nodes know they won't be doing endless work. It's up to the client to chunk the work up into reasonably large pieces (e.g. 12-24 hours of compute).
- Compute nodes do the work, encrypt the (hash of the output + the compute node's id aka wallet address) with the client's public key, and publish this. Only the client can therefore read the hashes. The client can verify that the hashes are unique per compute node because they contain the compute node's id.
- Clients mutate the job to set concurrency = 3 for 3% of the N pieces of work (or over time, roll a dice and do this for 3% of the jobs they submit). **The compute nodes do not know ahead of time whether a job is going to be verified or not**. It will be a surprise, this ensures they cannot only cheat the non-verified jobs. Requestor nodes may also need to spend a bit more money on the verification jobs, to incentivise nodes that do not have a local copy of

  the data to download it from IPFS - so the bid might need to be increased. Bid and concurrency are the only two mutable fields in a job spec.
- Clients require that the nodes that do the verification jobs are evenly distributed from the set of compute nodes. This is so that a malicious minority of compute nodes cannot hog all the verification jobs in aggregate. Verification jobs are worth more, so it's likely that lots of the compute nodes will vie for it; therefore the client should have the liberty to select from a majority pool of nodes.
    - If clients don't verify, they immediately pay out.
    - If clients do verify, and all three hashes match, they pay out
    - If clients do verify a job, and one hash doesn't match, they pay the two nodes that agree and slash the node that differs (or at least, don’t pay it).
    - If all three hashes don't match, they fail the job and funds get refunded to the client. This should be rare in practice assuming a majority of nodes in the network are honest (since they are incentivised to be honest).
- Compute nodes may note that they correctly ran the work and yet some client didn't pay out. If all 3 compute nodes agree that a client isn't behaving, they can gang up on the client and force payment of the work. The client wallet then gets slashed and a fixed amount of their funds burned. Clients must maintain a minimum wallet balance of this amount.
    - The servers gang up on the client by decrypting and publishing the hashes of the job they created. Then the whole world can see that they did the work and that they agree but that the client deemed them not to agree. This exact job will never be runnable on the network again though, since the hash of that work on that data hash(F(D)) is now public.
- Clients can only mutate the bid and concurrency of a job zero or one times, to avoid them constantly adjusting these values.
- To avoid malicious clients or servers just leaving things hanging forever, each step in the above protocol has a liveness constraint. If the next peer in each step fails to proceed in N blocks (or based on the block timestamps), the funds are refunded to the client or force-paid to the servers respectively.

It seems that the above does NOT require a reputation system. However further research, investigation and testing of the protocol is required to establish that for certain.

Reputation system

The system MAY additionally maintain a public reputation system based on the outcome of the protocol above. However, decisions will not be made programmatically based on the reputation system.

# Partial verification

The goal of partial verification is to avoid having to re-run every job 3 times to have any confidence in the veracity of the result. Failing to have this would get us branded with a "not eco friendly" brush. I call this the 103% cost of using the network, i.e. can you have a 3% overhead rather than a 200% overhead? If you have the incentive structure set up appropriately (see above) so that it's not economically rational for participants to lie, then the clients can only check a fraction of the work and still be confident in the results.

This is similar conceptually to how train companies don't have to check 100% of the tickets. If they just check your ticket 3% of the time, but the fine for not having a ticket is 1000x bigger than the cost of the tickets, then it's economically rational to buy a ticket every time.

The protocol above has this property. Clients can efficiently check just 3% of the results and still have confidence that the results are highly likely to be accurate because the servers didn't know at the time that they were doing the work whether they would be checked or not.

# Nondeterministic jobs such as ML training

Significant effort during the prototyping phase went into investigating support for nondeterministic workloads. An alternative approach to simply comparing output hashes – nondeterministic tracing – i.e. looking at the CPU and memory profiles of the work being done.

Links:

The concept, capturing the traces:
https://github.com/filecoin-project/bacalhau/wiki/Bacalhau-project-report-20220128

Calculating the traces:
https://github.com/filecoin-project/bacalhau/wiki/Bacalhau-project-report-20220211

Clustering the traces and giving results a tick or cross:
https://github.com/filecoin-project/bacalhau/wiki/Bacalhau-project-report-20220311

However in the down-scoping of the production project, we've decided to only support deterministic workloads initially. Supporting nondeterministic workloads requires significant signal processing & ML work to do properly, and even then the system has tolerances in it which mean that it will be more likely to be exploitable. It’s doable, but will have to be done after the initial release.

I still believe that supporting nondeterministic workloads - and thus generic container images rather than constrained deterministic WASM only - is essential for Bacalhau to get widely adopted as a compute substrate. Let’s see how far we get with the deterministic initial production release, and scope in adding support for this in a later project.