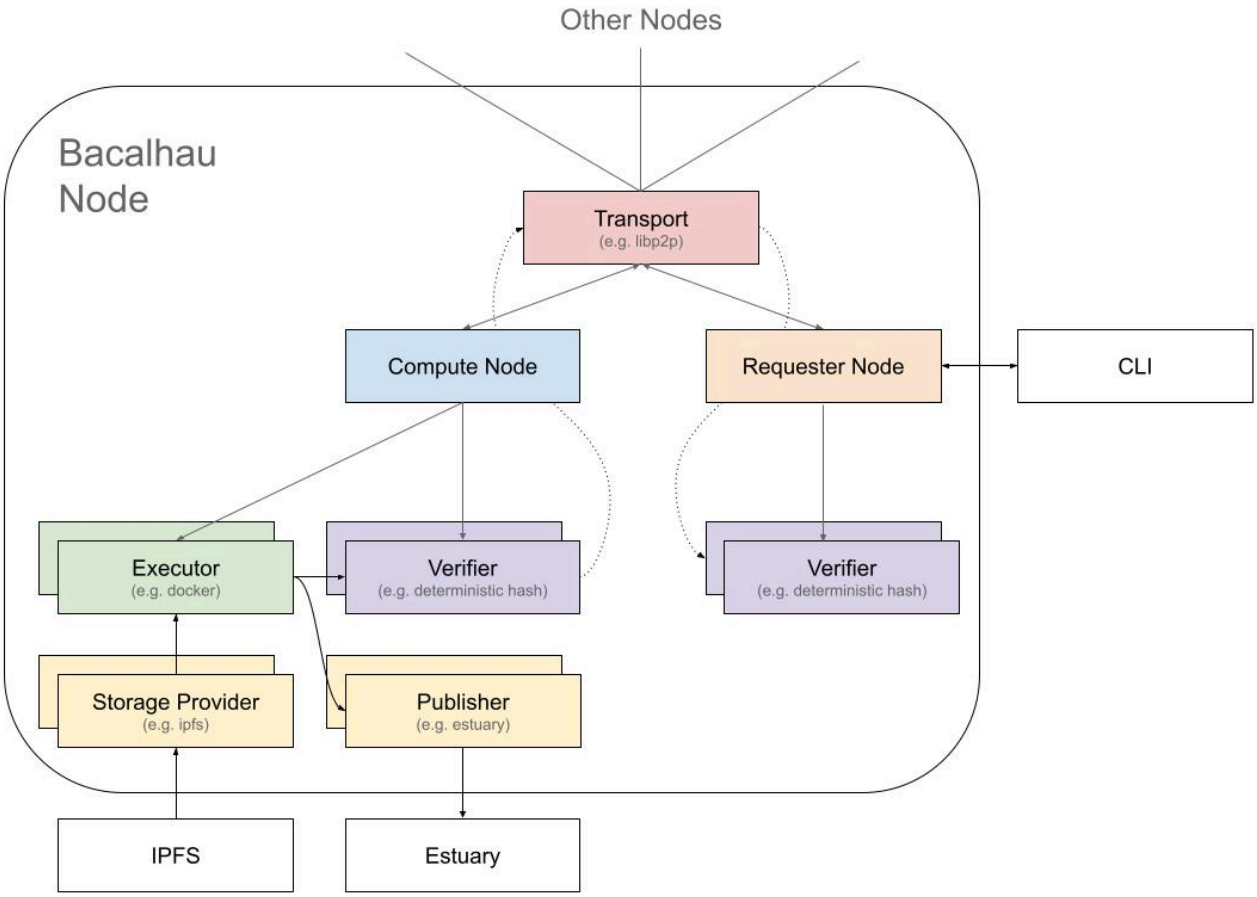
Architecture

Bacalhau is a peer-to-peer network of nodes that allows for decentralized communication between computers. Each node in the network has two components: a **requestor** component and a **compute** component.



To interact with the Bacalhau network, users can use the Bacalhau CLI (command-line interface) to send requests to a node in the network. These requests are sent using the JSON format over HTTP, a widely-used protocol for transmitting data over the internet.

When a request is received by a node in the network, it is broadcasted over the transport layer to all other nodes in the network. The transport layer is responsible for transmitting data between nodes, and because all nodes are connected to it, they all have a shared view of the network.

This shared view enables efficient and decentralized communication between nodes in the network, as all nodes are immediately made aware of any updates or changes made by other nodes.

System Components

Bacalhau's architecture involves to main sections which is the **core components** and **interfaces**.

Core Components

The core components are responsible for handling requests and connecting different nodes. It consist of:

- Requester node
- Compute node

Requester node

In the Bacalhau network, the requestor node is responsible for handling requests from clients using JSON over HTTP. This node serves as the main custodian of jobs that are submitted to it.

When a job is submitted to a requestor node, it broadcasts the job to the network and accepts or rejects the bids that come back for that job. It is important to note that there is only ever a single requestor node for a given job, which is the node that the job was originally submitted to.

Once the compute nodes in the network have executed the job, they will produce verification proposals. These proposals will be collated and combined by the requestor node when enough have been proposed. At this point, the proposals will be either accepted or rejected, and the compute nodes will then publish their raw results.

Overall, the requestor node plays a crucial role in the Bacalhau network, serving as the main point of contact for clients and the primary handler of jobs that are submitted to the network. By effectively managing job requests and verification proposals, the requestor node enables efficient and effective communication between nodes in the network, making it a valuable tool for decentralized computing applications.

Compute node

In the Bacalhau network, the compute node plays a critical role in the process of executing jobs and producing results. When a new job is detected on the network, the compute node will determine whether it wants to bid on that job or not. If a bid is made and accepted by the requester node, the compute node will run the job using its collection of executors, each of which has its own collection of storage providers.

Once the executor has run the job and produced results, the compute node will generate a verification proposal. This proposal will be collated by the requester node along with proposals from other compute nodes that ran the same job. These proposals will then be either accepted or rejected, after which the compute node will publish its raw results via the publisher interface. The compute node has a collection of named executors, verifiers, and publishers, and it will choose the most appropriate ones based on the job specifications.

Overall, the compute node is a crucial component of the Bacalhau network, responsible for executing jobs and producing results. By efficiently managing the bidding and verification process, the Compute node enables effective communication between nodes in the network, making it a valuable tool for decentralized computing applications.

Interface

The interface handles the distribution, execution, storage, verification and publishing of jobs.

- Transport
- Executor
- Storage Provider
- Verifier
- Publisher

Transport

The transport interface is responsible for sending messages about jobs that are created, bid upon, and executed to other compute nodes. It also manages the identity of individual Bacalhau nodes to ensure that messages are only delivered to authorized nodes, which improves network security.

To achieve this, the transport interface uses a protocol called **bprotocol**, which is a point-to-point scheduling protocol that runs over libp2p and is used to distribute job messages

efficiently to other nodes on the network. This is our upgrade to the GossipSub handler as it ensures that messages are delivered to the right nodes without causing network congestion, thereby making communication between nodes more scalable and efficient.

Executor

The executor is a critical component of the Bacalhau network that handles the execution of jobs and ensures that the storage used by the job is local. One of its main responsibilities is to present the input and output storage volumes into the job when it is run.

It is important to note that storage can differ between Docker and WebAssembly (wasm). Therefore, if a job references a specific CID (Content IDentifier), two different storage providers may be used based on the executor used for the job.

The executor performs two primary functions:

- presenting the storage volumes in a format that is suitable for the executor, and,
- running the job.

When the job is completed, the executor will merge the stdout, stderr, and named output volumes into a results folder. This results folder is used to generate a verification proposal that is sent to the requester.

Once the proposal is accepted or rejected, the results folder is then forwarded to the publisher for publication. Overall, the executor plays a crucial role in the Bacalhau network by ensuring that jobs are executed properly, and their results are published accurately.

Storage Provider

In a peer-to-peer network like Bacalhau, storage providers play a crucial role in presenting an upstream storage source. There can be different storage providers available in the network, each with its own way of manifesting the CI to the executor.

For instance, there can be a posix storage provider that presents the CID as a POSIX filesystem, or a library storage provider that streams the contents of the CID (Content IDentifier) via a library call.

On the other hand, there can be different executor implementations, such as Docker or WASM, that are responsible for running the job. When a job is submitted to these executors, the executor should select the appropriate storage provider to work with, depending on the implementation.

For instance, the Docker executor might use the posix storage provider to manifest the CID as a POSIX filesystem, while the WASM executor might use a library storage provider to stream the contents of the CID (Content IDentifier) via a library call.

Therefore, the storage providers and Executor implementations are loosely coupled, allowing the POSIX and library storage providers to be used across multiple executors, wherever it is deemed appropriate.

Verifier

The verifier component is responsible for ensuring the validity of the results produced by the executor and transporting those results back to the requester node. There are two main tasks that the verifier performs:

- checking the results produced by the executor against results produced by other nodes, and,
- transporting those results back to the requester node.

How the results are checked depends on the nature of the job. For example, if the job is deterministic, the "deterministic hash" verifier can be used to ensure that the results produced by different nodes are the same. However, if the job is non-deterministic, a different approach must be used to check the results.

Both the compute node and the requester node have a verifier component. The compute node verifier produces a verification proposal based on having run the job, while the requester node verifier collates the proposals from various compute nodes. Once enough proposals have arrived, the requester node verifier decides which compute nodes have actually performed the work and emits events to the network indicating whether the verification proposals are accepted or rejected.

Publisher

The publisher is responsible for uploading the final results of a job to a public location where clients can access them. However, before the results can be published, the verification process must be completed to ensure the accuracy and authenticity of the results.

During the verification process, the raw results are kept private between the compute node and requester node to prevent unauthorized access or copying. Once the verification is complete and the results are deemed valid, the publisher interface takes over the task of

uploading the local folder of results to a public location that can be accessed by the wider network.

The default publisher used is either Estuary, with the published results being stored with a unique content identifier (cid) that can be used to retrieve them. If Estuary is used as the publisher, the results will also be stored on Filecoin. The publisher interface is responsible for ensuring that the published results are accessible and can be read by other nodes on the network.

Job Lifecycle

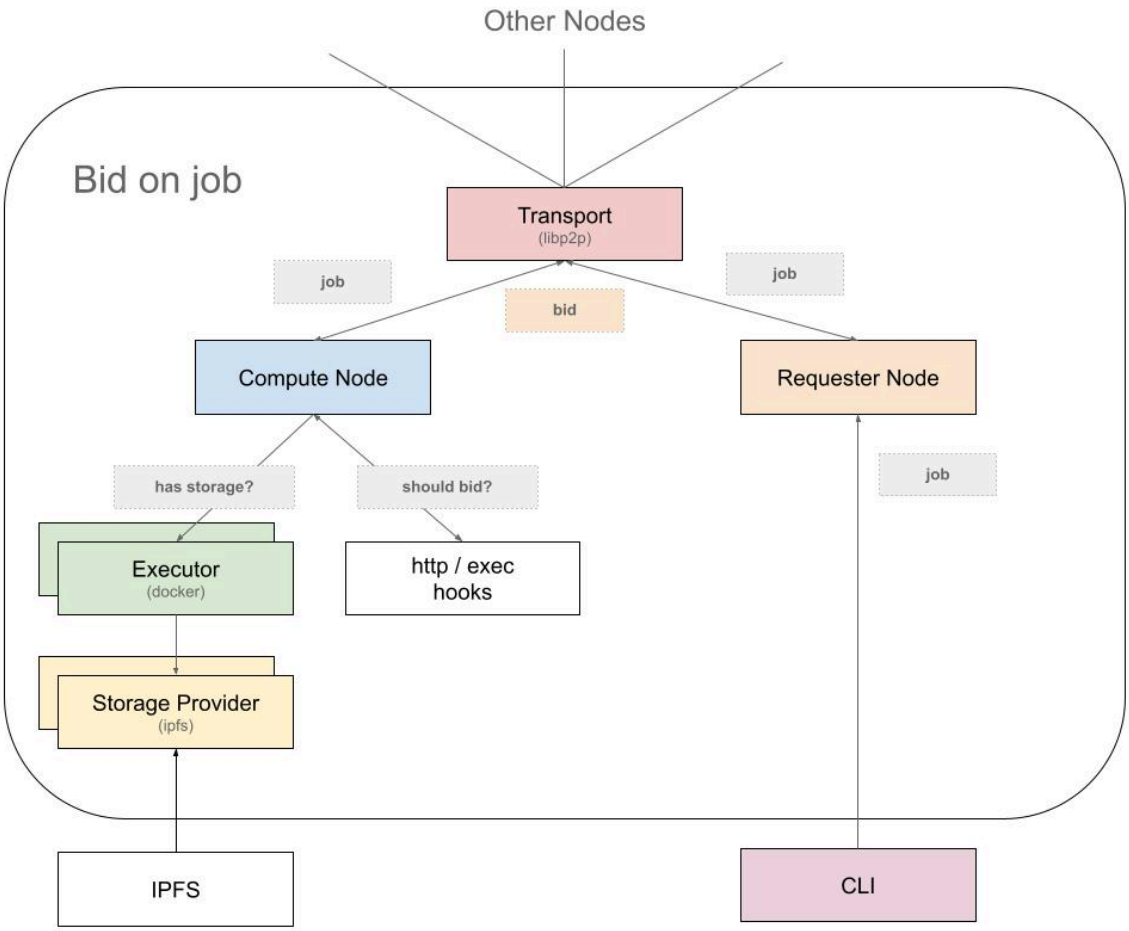
The job lifecycle involves several steps that are handled by different components of the network, from job submission to jobs download.

Job Submission

Jobs submitted via the Bacalhau CLI are forwarded to a Bacalhau network node at `bootstrap.production.bacalhau.org` via port 1234 by default. This Bacalhau node will act as the requestor node for the duration of the job lifecycle. Jobs can also be submitted to any requestor node on the Bacalhau network.

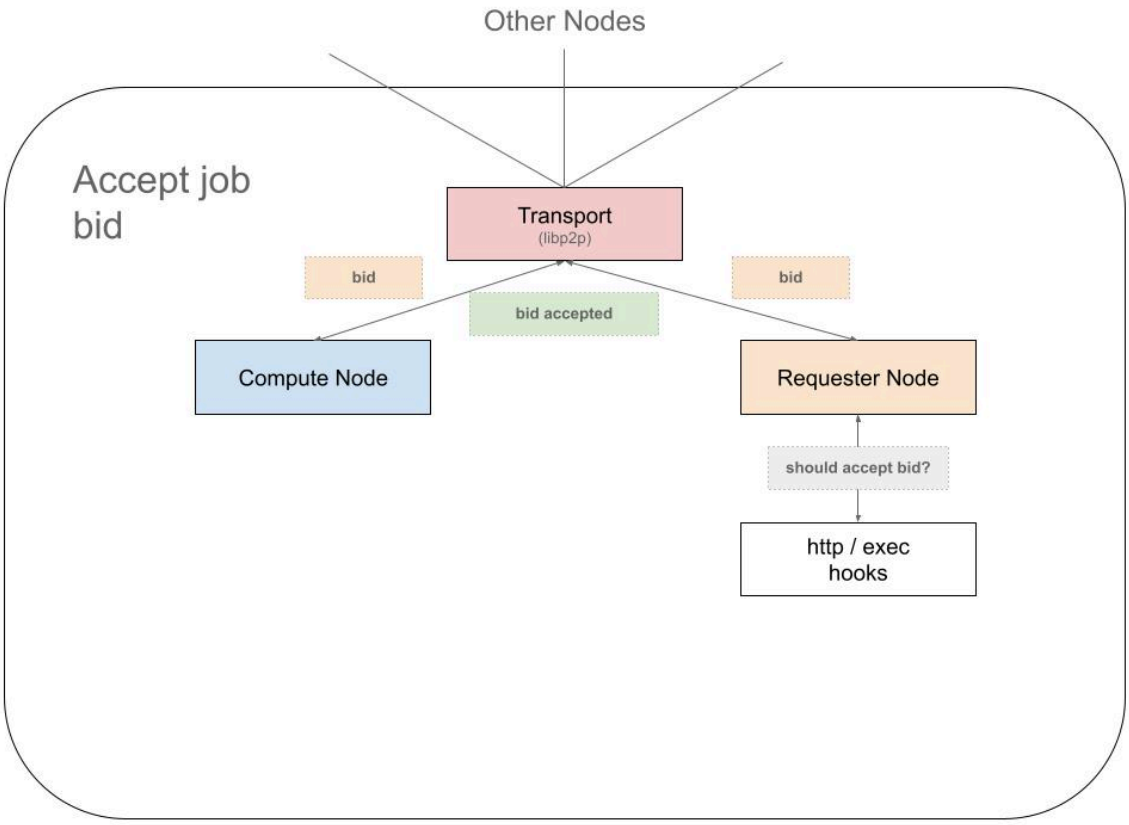
When jobs are submitted to the requestor node, all compute nodes hear of this new job and can choose to `bid` on it. The job deal will have a `concurrency` setting, which refers to how many different nodes you may want to run this job. It will also have `confidence` and `min-bids` properties. Confidence is how many verification proposals must agree for the job to be deemed successful. `Min-bids` is how many bids must have been made before we will choose to accept any.

The job might also mention the use of `volumes` (for example some CIDs). The compute node can choose to bid on the job if the data for the volumes resides locally to the compute node, or it can choose to bid anyway. Bacalhau supports the use of external http or exec hooks to decide if a node wants to bid on a job. This means that a node operator can give granular rules about the jobs they are willing to run.



Job Acceptance

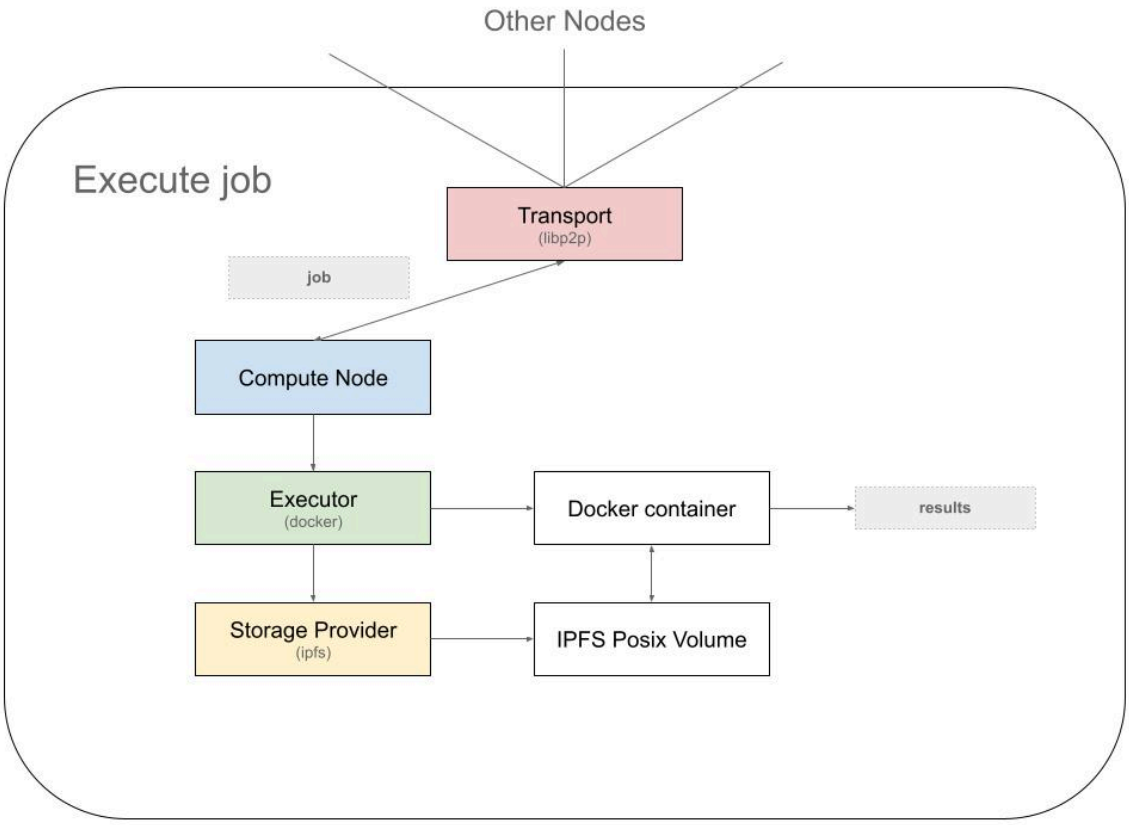
As bids from compute nodes arrive back at the originating requester node, it can choose which bids to accept and which ones to reject. This can be based on the previous reputation of each compute node, or any other factors the requestor node might take into account (like locality, hardware resources, cost etc). The requestor node will also have the same http or exec hooks to decide if it wants to accept a bid from a given compute node. The `min-bids` setting is useful to ensure that we don't accept bids on a first bid first accepted basis.



Job Execution

As accepted bids are received by compute nodes, they will `execute` the job using the executor for that job, and the storage providers that executor has mapped in.

For example, a job could use the `docker` executor, `WASM` executor or a library storage volumes. This would result in a POSIX mount of the storage into a running container or a WASM style `syscall` to stream the storage bytes into the WASM runtime. Each executor will deal with storage in a different way, so even though each job mentions the storage volumes, they would both end up with different implementations at runtime.



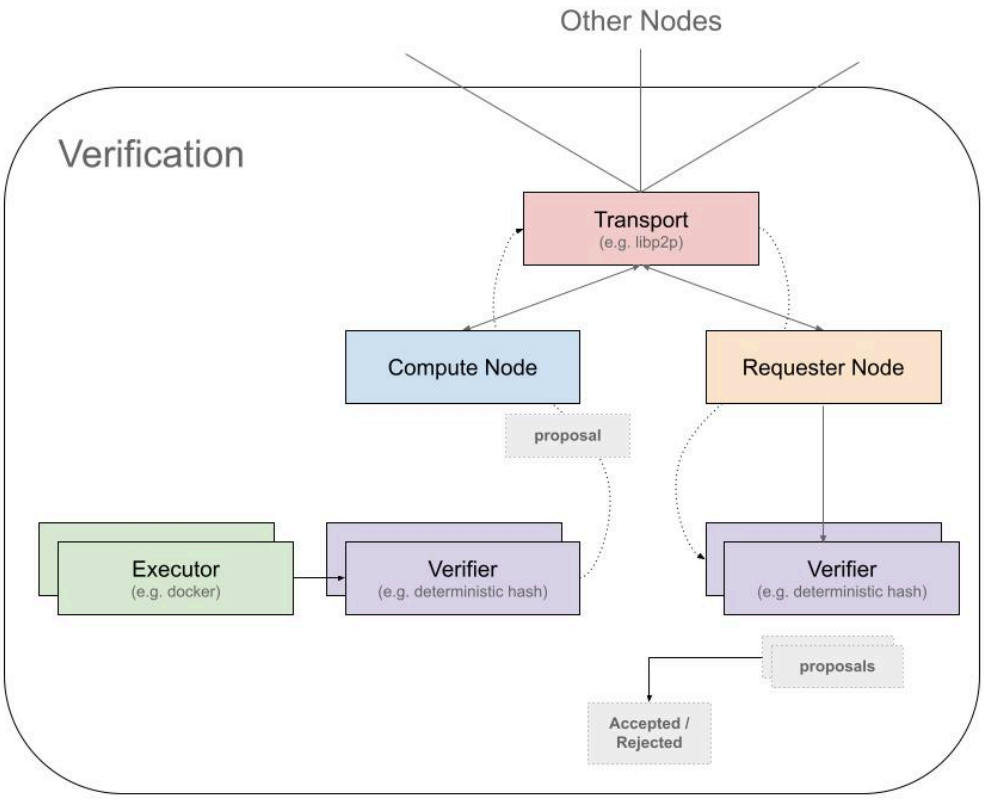
Verification

Once the executor has completed the running of the job, a verification proposal will be generated by the verifier module running on the compute node. The nature of this proposal depends on the module used. For example, the deterministic hash verifier will:

- Calculate a `sha256` hash of the contents of the results folder
- Encrypt this hash using the public key of the requester node
- Broadcast the encrypted hash over the network
  - Nodes that are NOT the requester node cannot copy the hash because they do not have the requesters private key
  - The requester will use it's private key to decrypt the message and read the hash
  - This means that bad actors cannot simply copy the results hash from other nodes
- The requester node will wait for enough proposals before comparing the results hashes
- It will then broadcast "results accepted" and "results rejected" events based on it's decision for verification

In the case of a deterministic job, a user can guard against malicious compute nodes that join the network, bids and propose wrong results by using `--verifier deterministic --min-bids N --concurrency 3` (where N is, say, >half of the size of the network, currently 9). This will require that *N* bids are received before the requestor node chooses between them randomly. So when you submit WASM jobs (which are deterministic) this can give you a good level of confidence the jobs are evenly spread across nodes and malicious nodes will be, on average, caught out.

It's possible to use other types of verification methods by re-implementing the verification interface and using another technique.

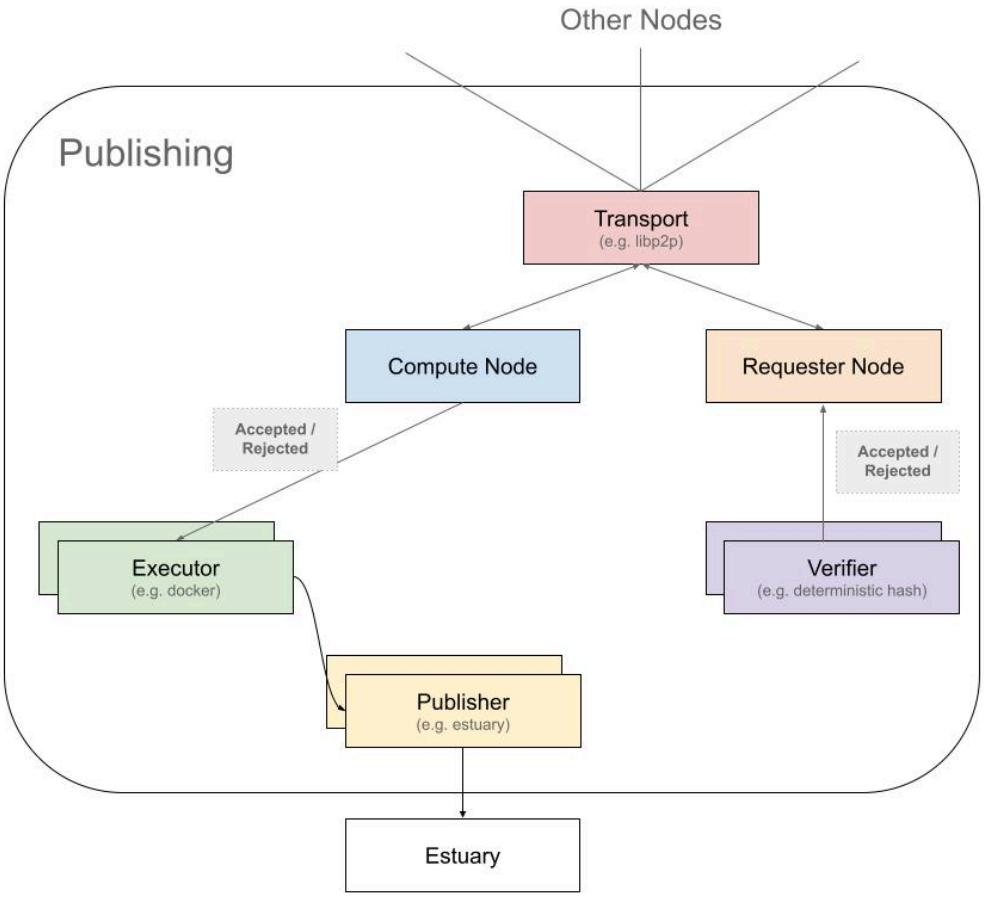


Publishing

Once verification has resulted in `results accepted` or `results rejected` events, the publisher will publish the raw results folder currently residing on the compute node.

The default publisher is `Estuary` (if no API key is provided this falls back to the IPFS publisher). The publisher interface mainly consists of a single function, which has the task of

uploading the local results folder somewhere and returning a storage reference to where it has been uploaded.



Networking

It is possible to run Bacalhau completely disconnected from the main Bacalhau network, so that you can run private workloads without risking running on public nodes or inadvertently sharing your data outside of your organization. The isolated network will not connect to the public Bacalhau network nor connect to a public network. Read more on networking

Input / Output Volumes

A job includes the concept of input and output volumes, and the Docker executor implements support for these. This means you can specify your CIDs, URLs, as input paths and also write results to an output volume. This can be seen by the following example:

```
cid=$(ipfs add file.txt)
bacalhau docker run \
  -i ipfs://$cid:/file.txt \
```

```
  -o apples:/output_folder \
  ubuntu \
  bash -c 'cat /file.txt > /output_folder/file.txt'
```

The above example demonstrates an input volume flag `-i ipfs://$cid:/file.txt`, which mounts the contents of $cid within the docker container at location `/file.txt` (root).

Output volumes are mounted to the Docker container at the location specified. In the example above, any content written to `/output_folder` will be made available within the apples folder in the job results CID.

Once the job has run on the executor, the contents of `stdout` and `stderr` will be added to any named output volumes the job has used (in this case apples), and all those entities will be packaged into the results folder which is then published to your storage location via the verifier.