Computes Mesh: Computing for the Masses

Computes, inc.

February 2018 *

Abstract

Around the world, millions of CPU and GPUs are sitting idle. Many organizations have tens of thousands of workstations, racks filled with servers, idling during off-peak hours. Artificial intelligence, machine learning, blockchains, and other technologies are increasing demand for computing resources at an unprecedented rate. The burdens of massive parallel processing are being placed on organizations large and small. What would happen if organizations could utilize this latent computing power to begin to address need for massive computational power? What if that work could be done while minimizing allocation of new resources?

Computes Mesh is a *distributed, decentralized mesh computer* built on top of IPFS[?] . The Computes Mesh is fault tolerant, efficient, and intelligent. Members of the Mesh are heterogeneous, consisting of different CPU frequencies, operating systems, differing amounts of RAM and storage, even different CPU architectures.

This paper will show how the core components of Computes Mesh work together to create a mesh computer. Detailed explainations of each component can be found in their individual whitepapers.

*Computes Mesh is a work in progress. New versions of this paper will appear at `https://www.computes.com/whitepapers`. For comments or corrections, contact us at `whitepaper@computes.com`.

Contents

List of Figures

Listings

1 Introduction

The Computes Mesh is designed to allow for the distribution of small, parallelizable tasks across a disparate array of physical computing devices. The devices may be heterogenous, made up of different operating systems, CPU architectures, power requirements, and availability.

1.1 Components

Computes Mesh is constructed using IPFS, the Lattice protocol, and a set of core subsystems. The subsystems can be mixed and matched across all members in the Computes Mesh, allowing users to determine how best to use their resources.

1. **??**: A decentralized, distributed datastore used by all subsystems of the Computes Mesh.
2. **??**: A work queue manager using Lattice to safely and reliably distribute work across the Computes Mesh.
3. **??**: An observer of the Lattice protocol that allows for long term data storage and retreival. This allows most members in the system to retain only the data necessary to process the work at hand.
4. **??**: A framework for secure, reliable, distributed computation across a heterogenous network of computers.
5. **??**: A rule based system for chaining tasks together based on changes made to results or work status.
6. **??**: A layer on top of the Manager that allows for checking results and building trust across the Computes Mesh.

1.2 Key Terms

There are some key terms that are used in the examples:

- Task: A unit of work that is placed in the Queue and executed by the Runner. The exact schema of a Task is out of scope for this document.
- Engine: An authorized binary used by the **??** that executes the payload defined in the Task.
- HPCP: The Hash Pointer Collaboration Protocol, defined in Lattice[?] . Allows clients to share the latest version of a datastructure.
- Queue: A Grow Only Hash Object[?] that defines a distributed, decentralized work queue. Used by the Manager to assign, execute, and complete Tasks.
- Condition: A simple rule run by the **??** that when met allows a new Task to be enqueued based on the results of a previous Task.
- Results: A Grow Only Hash Object[?] that contains the results of one or more tasks.

2 Lattice

Please see the Lattice whitepaper[?] for details.

3 Manager

The Manager is responsible for Task Management, including enqueueing new tasks, handing the next available task to the **??**, and completing tasks. The Manager uses **??** extensively to manage the Queue, Status, and Results.

3.1 Manager Protocol

(Assign, Complete, Enqueue, Next, Status)

- Assign(task): Clients execute the Assign protocol to *add* the task to the set of assigned tasks in the Queue.

- Complete(task): Clients execute the Complete protocol to *add* the task to the set of completed tasks in the Queue.
- Enqueue(data): Clients execute the Enqueue protocol to *add* `data` to the set of available tasks in the Queue.
- Next() → data: Clients execute the Next protocol to *get* the next available Task. If there are no more tasks available, nil may be returned. The next task is a Task that is in the available set, but not found in the assigned or completed set.
- Status(taskHash) → data: Clients execute the Status protocol to *get* the GOHO that represents the current task status.

3.2 Distributed Queue

Utilizing the core components of Lattice, a distributed, decentralized work queue can be assembled.

3.2.1 Participants

All actors participate as a Client. Clients modify Grow Only Hash Objects and communicate changes to other Clients using the Hash Pointer Collaboration Protocol.

3.2.2 Queue

The Queue is a Grow Only Hash Object that contains three grow-only sets: available, assigned, and completed. The origin hash of the Queue is known to all Clients ahead of time and used as the basis of HPCP communication. The availability of a work item is determined by its absence from the assigned and completed sets.

3.2.3 Work Queue Cycle

An informal overview of the Work Queue Cycle

- Client C_1 appends a work item to the available set in the Queue.

$$Q_0.\texttt{Append}(\texttt{available}, W_1) \rightarrow Q_1$$

$$\texttt{HPCP.Update}(Q_0, Q_1)$$

- Client C_2 receives HPCP update, dequeues work item and marks it as assigned.

$$Q_1.\texttt{Append}(\texttt{assigned}, W_1) \rightarrow Q_2$$

$$\texttt{HPCP.Update}(Q_0, Q_2)$$

- Client C_2 completes work item.

$$Q_2.\texttt{Append}(\texttt{completed}, W_1) \rightarrow Q_3$$

$$\texttt{HPCP.Update}(Q_0, Q_3)$$

3.3 Example Queue

Assuming a client appends 3 different work items into the available queue:

$$Q_0.\texttt{Append}(\texttt{completed}, \texttt{abc...}) \rightarrow Q_1$$

$$Q_1.\texttt{Append}(\texttt{completed}, \texttt{def...}) \rightarrow Q_2$$

$$Q_2.\texttt{Append}(\texttt{completed}, \texttt{ghi...}) \rightarrow Q_3$$

Listing 1: Queue Example JSON

```
{
  "available": [
     "abc...",
     "def...",
     "ghi..."
  ]
```

```
}
```

3.4 Constraints

Constraints are a way for a task to indicate basic requirements, such as GPU, minimum CPU clock speed, CPU architecture, etc.

TODO: Example

4 Archivist

The Archivist is an observer of Lattice and will make any part of the system available for later retrieval. This allows network designers to designate members that have the appropriate levels of storage available to be Archivists, freeing other members to focus on computation without needing access to the full datasets.

4.1 Archivist Protocol

(Pin)

- Pin(hash, [genisys]): Clients execute the Pin protocol to *update* the IPFS pin that will allow content to be distributed on an as-needed basis to other Clients. The previous version of a GOHO (if genisys is provided) will be unpinned to optimize storage.

5 Runner

The Runner downloads the Engine and uses it to execute the task. An Engine is a secure, pre-verified binary, authorized by Computes.

5.1 Runner Protocol

(Download, Run)

- Download(task.engine): Clients execute the Download protocol to *retrieve* the Engine defined in the Task.

- Run(task): Clients execute the Run protocol to *execute* the Engine binary passing in the Input defined in the Task. The results of Run are added to the Results GOHO.

5.2 Runner Example

In this case, the manager looks up the hash that matches the engine `docker-json-runner`, the engine is downloaded via IPFS and executed, passing `metadata` and `input` to the engine. On engine exit, the output will be set on the `result` datastore.

Listing 2: Example Task JSON

```
{
  "input": {
    "dataset": { "/": "zdpu..." },
    "path": "split/input"
  },
  "taskDefinition": {
    "runner": {
      "type": "docker-json-runner",
      "metadata": {
        "image": "computes/fibonacci-sum-split:latest"
      }
    },
    "result": {
      "action": "set",
      "destination": {
        "dataset": { "/": "zdpu..." },
        "path": "split/results"
      }
    }
  }
}
```

6 Evaluator

The Evaluator uses the **??** to allow tasks to be chained together. Using a simple rule language, Conditions are specified in a task and, if met, will enqueue one or more new tasks into the Queue.

6.1 Decentralized Reactive Event System Protocol

The Decentralized Reactive Event System (DRES) allows Clients to evaluate a set of rules and insert new items into a Grow Only Hash Object. This may trigger subsequent events and can be used to continually insert new work into Lattice without performing additional work.

The DRES allows any member of the Computes Mesh to check the conditions, even if the member was not involved in the task. This prevents deadlock situations where clients are simultaneously adding results but have not yet seen the updates from one another. Any other member that merges their changes will be able to evaluate the conditions and enqueue tasks if necessary.

Due to the content addressable nature of IPFS, if multiple members respond to the same event, they will enqueue the same task hash and will not conflict with each other, or cause duplicate work to be performed.

(Evaluate)

- Evaluate(genesis) → [hash, ...]: Clients execute the Evaluate protocol and *may evaluate* a set of rules that produce a set of new changes to a Grow Only Hash Object.

6.2 Example Conditions

In this example, a simple condition has been created. The latest version of the dataset will be retrieved from IPFS, then the Evaluator will check to see that the data at the path `/map/results` is falsey.

If the condition is `true`, the Evaluator will map over the result path and enqueue a new task for every member of the set.

Listing 3: Example Conditions JSON

```
{
```

```
  "taskDefinition": {
    "conditions": [
      {
        "name": "Create Map tasks",
        "condition": "!exist(dataset(hpcp('zdpu.../map/results')))",
        "taskDefinition": {
          "/": "zdpu..."
        },
        "action": "map"
      }
    ]
  },
  "result": {
    "dataset": {
      "/": "zdpu..."
    },
    "path": "split/results"
  }
}
```

7 Verifier

The Verifier uses the Distributed Trust System[?] to validate tasks performed by other Clients. This protects the integrity of Lattice and the Computes Mesh. Clients that produce unreliable results will be prevented from running new tasks.

8 Task Flow Overview

This section will provide a simplified overview of a Task moving through the Computes Mesh. The purpose of this overview is to show how the core components of Computes work together, not to provide a precise view of the implementation.

In Figure **??**, a new Task is added to the Queue (via Lattice). When the Task is available, the Runner will download the appropriate Engine as specified in the Task. The Runner starts the downloaded Engine and passes in the Task information.

Any time a datastructure is changed, the new Merkle Root is published using HPCP. The Archivist observes and records all HPCP changes for use by other clients at a later time. The Archivist will also Pin[?] the object in IPFS.

Whenever a new Merkle Tree is created, usually when the Task updates the Results, the Evaluator will evaluate all the Conditions in the Task and generate new tasks if the conditions are met.

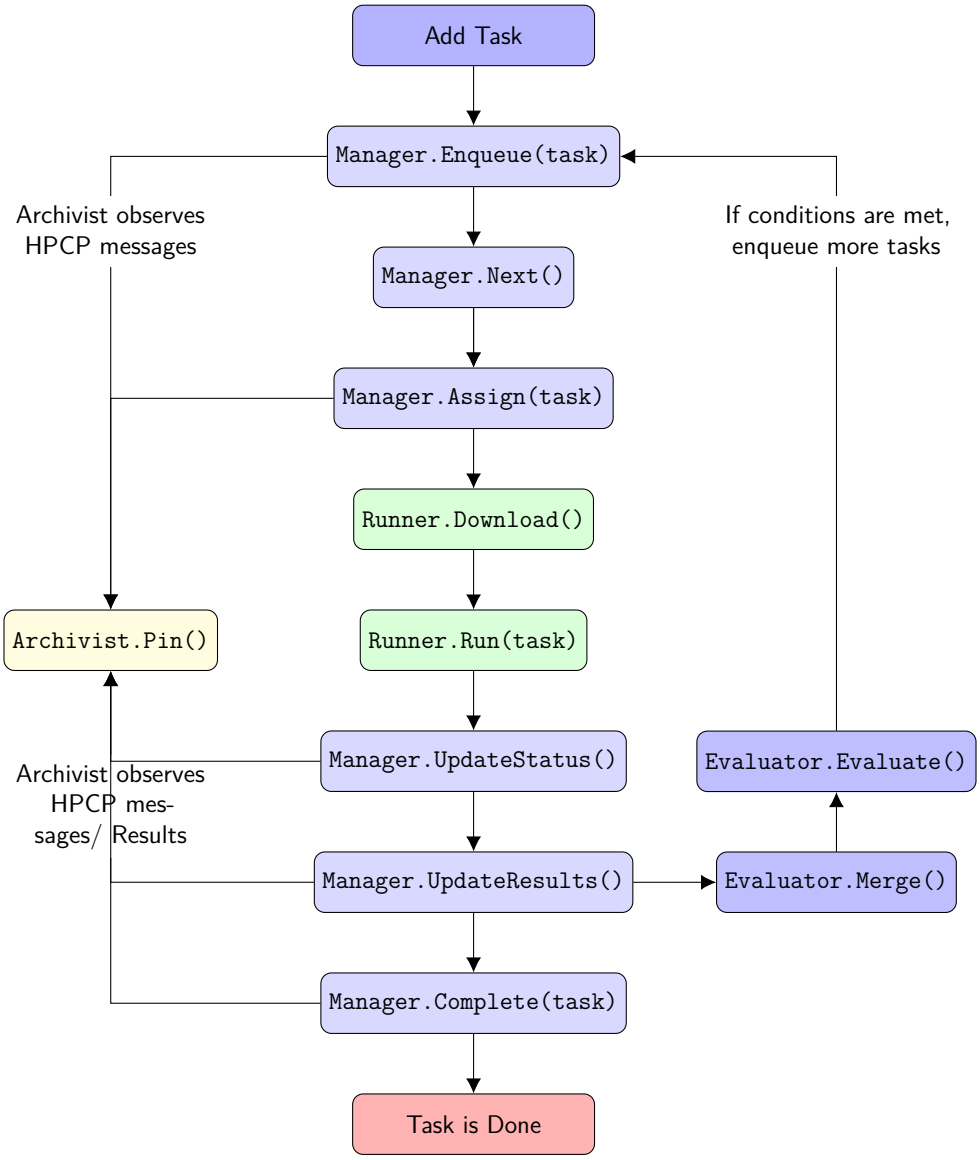

Figure 1: Simplified Task Flow