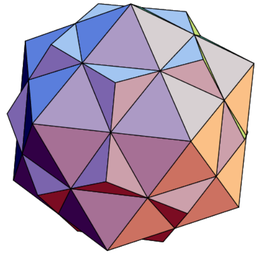
Coordinating web3 computing workloads

Mike Voronov
twitter.com/@vms11



fluence

Agenda

- Algorithms in distributed networks
- AquaVM: pi-calculus based approach
- Fluence Labs implementation
- Distributed Choreography and Composition Working Group (**DCC-WG**)

Algorithms
in distributed networks

Assumption & Question

Let's say we have a way to run a computation on a peer in the form of function and (possibly stateful and even effectful) services on peers:

$$\mathbf{f}_{peer} : \mathbf{CID} \rightarrow \mathbf{CID}$$

How to compose a set of such functions possibly located on different peers?

$$\mathbf{f}_{system} = (f_{peer_1} \circ ... \circ f_{peer_n}) : \mathbf{CID} \rightarrow \mathbf{CID}$$

Qs for $\mathbf{f}_{system}$

- How to put this function where it needs to be?
- How to find a peer ready to execute my code? Move data to code, code to data, etc?
- How to organize many peers to take their parts in a workflow?
- How to organize the network of different peers with different code there?
- How to protect from common attacks, like replay, Sybil and Eclipse?
- How to do all of this in the most efficient way, improve latency, etc?

Approaches

Centralized computing

- Middlemen (cloud, admin)
- No redundancy
- Unverifiable
- No consensus
- Cheap

Decentralized computing

✓ No middlemen

✓ No redundancy

✓ Verifiable (flexible)

✓ Optional consensus

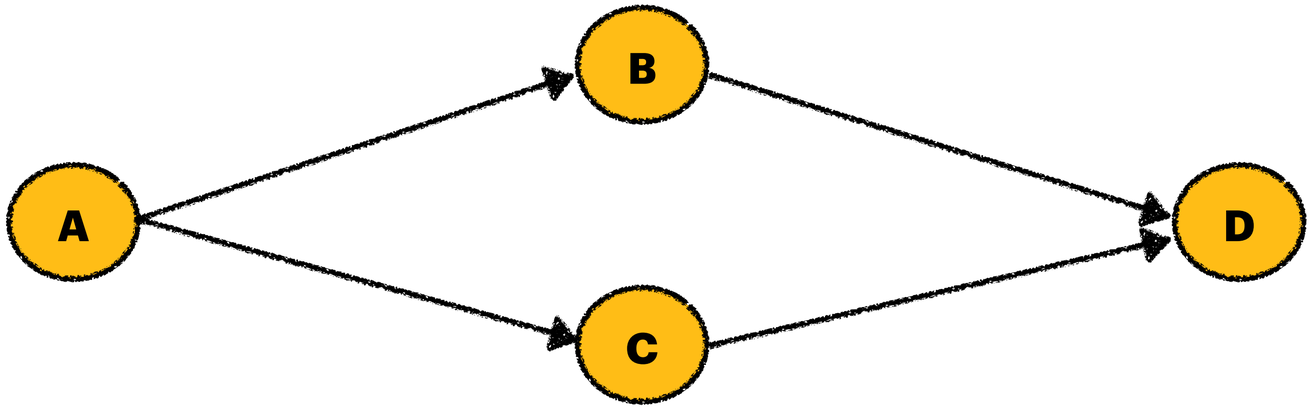
✓ Cheap

On-chain computing

- No middlemen
- Redundancy
- Verifiable
- BFT consensus
- Expensive

Observation

- there are some models to express and orchestrate decentralized computing, but many of them are expressed with the **fork-join pattern** (such as OpenMP, map-reduce and many others)



- is there any suitable theory for that?

pi-calculus to express fork-join

- Output, Input: a!b, a?b
- Combinators: **P.Q, P+Q, P|Q**
- Match, Mismatch: (x == y)P, (x != y)P
- Restriction: (new x)P
- Replication: !P := P | !P
- Null: .0

pi calculus

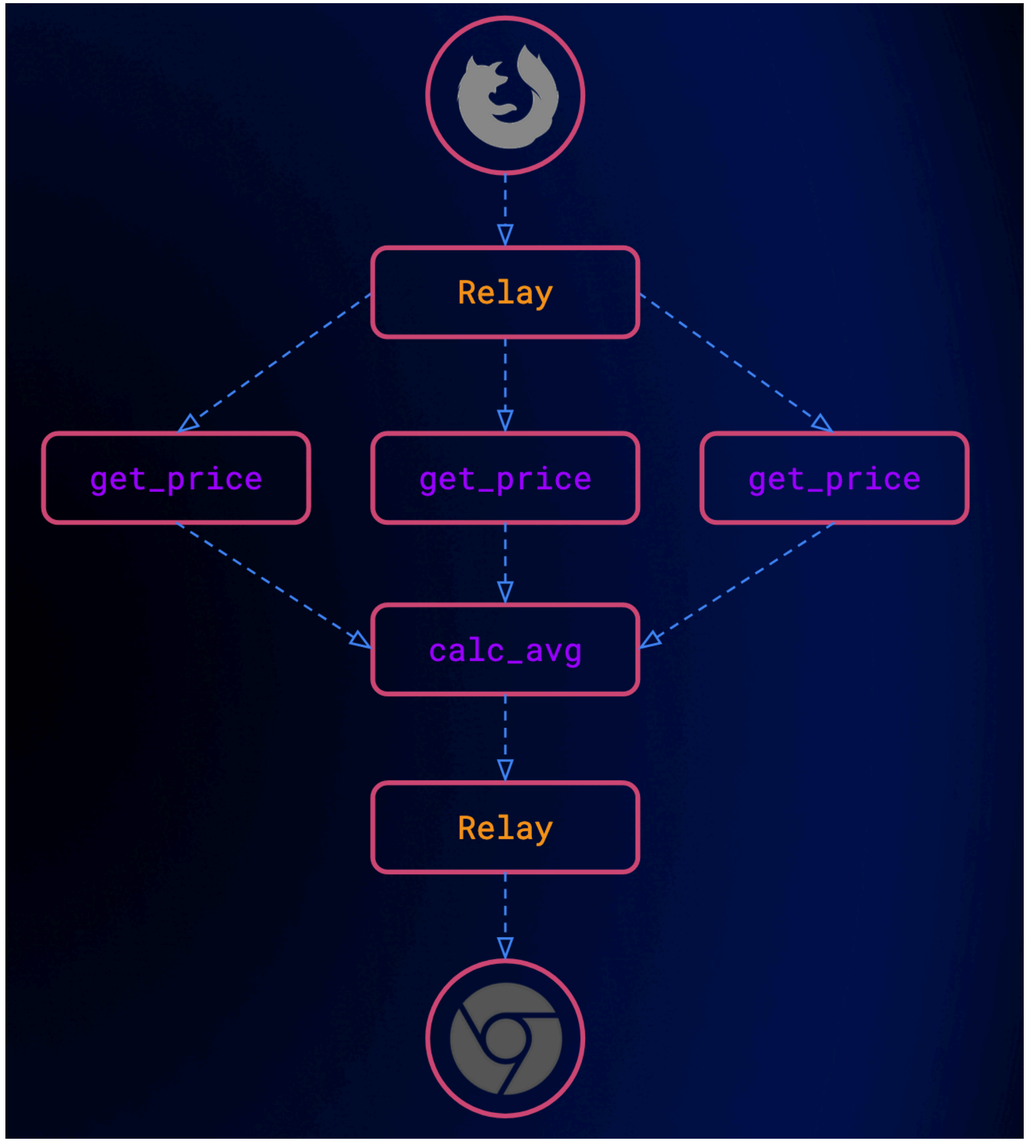
Aqua fork-join example

map_reduce.aqua

```
for p <- peers par:
   prices <- PriceService.get_price()

on avg_peer:
   avg <- AvgService.calc_avg(prices)

on received_peer via relay:
   ReceiveService.receive_avg(avg)
```



AquaVM: pi-calculus based approach

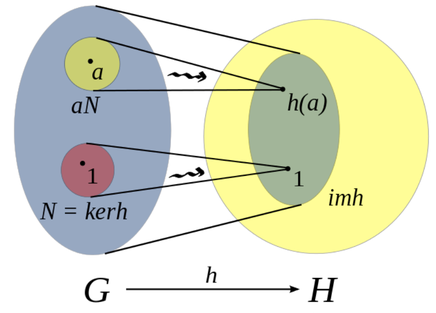
AquaVM's way to rule the complexity

To rule the complexity of development and verify our research ideas we use the idea of homomorphic mapping

f: X -> Y

where X is AquaVM

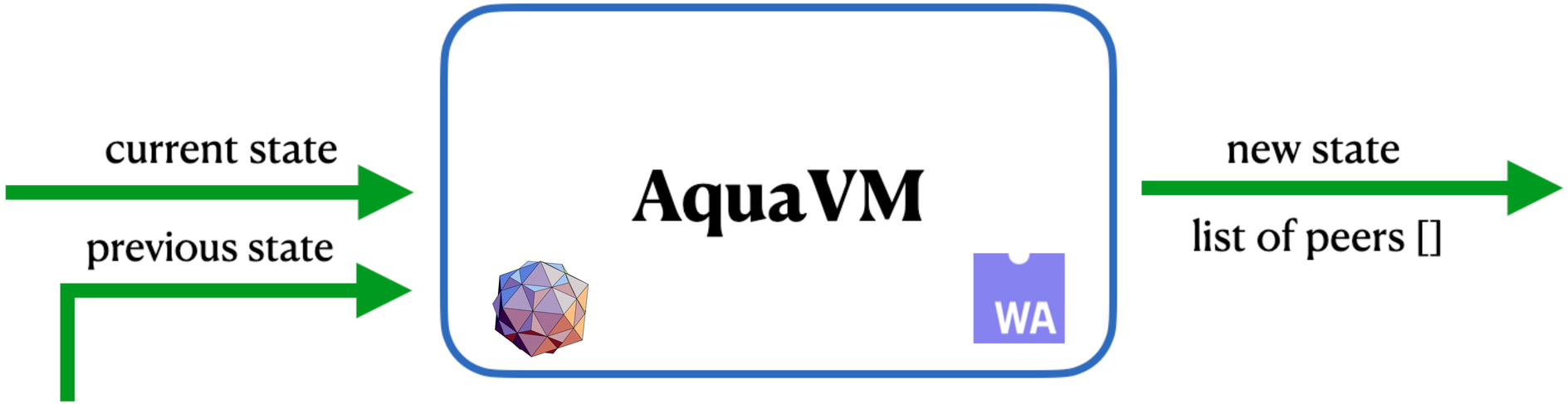
f(x + y) = f(x) * f(y)



What is Y?

- pi calculus
- lambda calculus
- abstract algebra
- category theory
- graph theory
- automata theory
- compiler theory
- TLA+

AquaVM as FSM



- AquaVM is a pure state transition function
- It takes previous state and state from a particle
  - returns a new state and a list of peers to send this state

automata theory

AquaVM as FSM

- AquaVM collects parameters of **service calls** that could be executed on this peer and return them after finishing execution
- then it expects to obtain **results** of their execution back

automata theory

AquaVM: triggers

AquaVM triggers:

- handle data from a new network packet (with the same particle_id)
- handle a call result from a called service

Pros of this scheme:

- interpreter is a pure function
- allows async/parallel service execution on a peer
- execution takes a fixed time

AquaVM instructions

call - call f_peer

pi calculus

AquaVM instructions

call - call f_peer

seq - sequential execution

par - parallel execution

xor - execution with catching errors

pi calculus

AquaVM instructions

call - call f_peer

match/mismatch
branching

new - create a new scope

ap - applying a functor

canon - canonicalization

null - does nothing

never - stops execution

fail - throws an error

seq - sequential execution

par - parallel execution

xor - execution with catching errors

fold/next - iteration

category theory

compiler theory

pi calculus

AquaVM: execution example

- every time execution starts from the very beggining of each script

- result state contains a lineralized execution trace

- let's consider how this simple fork-join example will be executed by AquaVM:

```
(seq
    (par
        (call "provider_1" ("prices" "get") [] price_1)
        (call "provider_2" ("prices" "get") [] price_2)
    )
    (seq
        (call "average" ("average_service" "calc") [price_1 price_2] avg_price)
        (call "user" ("recieve_service" "show") [avg_price])
    )
)
```

pi calculus

AquaVM: execution example

Let's say that we start execution on a peer with peer_id "user"

```
(seq
    (par
        (call "provider_1" ("prices" "get") [] price_1)
        (call "provider_2" ("prices" "get") [] price_2)
    )
    (seq
        (call "average" ("average_service" "calc") [price_1 price_2] avg_price)
        (call "user" ("recieve_service" "show") [avg_price])
    )
)
```

peer_id: "user"

seq takes two instructions and executes the second iff the first was executed

pi calculus

AquaVM: execution example

```
(seq
    (par                                                                    peer_id: "user"
        (call "provider_1" ("prices" "get") [] price_1)
        (call "provider_2" ("prices" "get") [] price_2)
    )
    (seq
        (call "average" ("average_service" "calc") [price_1 price_2] avg_price)
        (call "user" ("recieve_service" "show") [avg_price])
    )
)
```

par takes two instructions and executes them not depending on each other

pi calculus

AquaVM: execution example

AquaVM finishes with **next_peers = ["provider_1", "provider_2"]** and **new state**

```
(seq
    (par
        (call "provider_1" ("prices" "get") [] price_1)
        (call "provider_2" ("prices" "get") [] price_2)
    )
    (seq
        (call "average" ("average_service" "calc") [price_1 price_2] avg_price)
        (call "user" ("recieve_service" "show") [avg_price])
    )
)
```

peer_id: "user"

call executes function from a service on particular peer with the provided set of arguments bounding result to specified variable name

pi calculus

AquaVM: execution example

AquaVM finishes with **next_peers = ["average"]** and **new state**

```
(seq
    (par
        (call "provider_1" ("prices" "get") [] price_1)
        (call "provider_2" ("prices" "get") [] price_2)
    )
    (seq
        (call "average" ("average_service" "calc") [price_1 price_2] avg_price)
        (call "user" ("recieve_service" "show") [avg_price])
    )
)
```

peer_id: "provider_1", "provider_2"

call has join behaviour and waits for all their arguments to be defined

pi calculus

AquaVM: execution example

AquaVM finishes with **next_peers = ["user"]** and **new state**

```
(seq
    (par
        (call "provider_1" ("prices" "get") [] price_1)
        (call "provider_2" ("prices" "get") [] price_2)
    )
    (seq
        (call "average" ("average_service" "calc") [price_1 price_2] avg_price)
        (call "user" ("recieve_service" "show") [avg_price])
    )
)
```

peer_id: "average"

seq takes two instructions and executes the second iff the first was executed

pi calculus

AquaVM: execution example

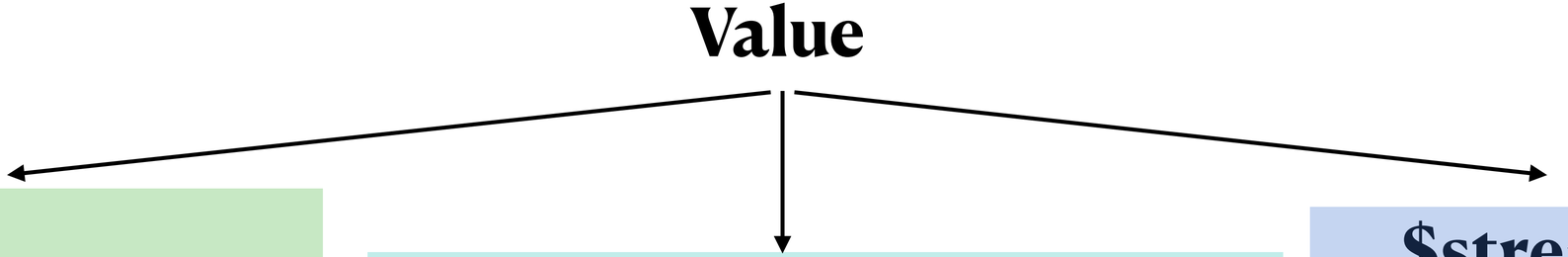
Finally, recieve_service.show is called with the result

```
(seq
    (par
        (call "provider_1" ("prices" "get") [] price_1)
        (call "provider_2" ("prices" "get") [] price_2)
    )
    (seq
        (call "average" ("average_service" "calc") [price_1 price_2] avg_price)
        (call "user" ("recieve_service" "show") [avg_price])
    )
)
```

peer_id: "user"

pi calculus

AquaVM: value types



scalars

- fully consistent
- could be an arg of any instruction
- JSON-like (at the moment)

canonicalized #$streams/#%maps

- create by fixing streams and maps on specified peer
- have mixed algebra

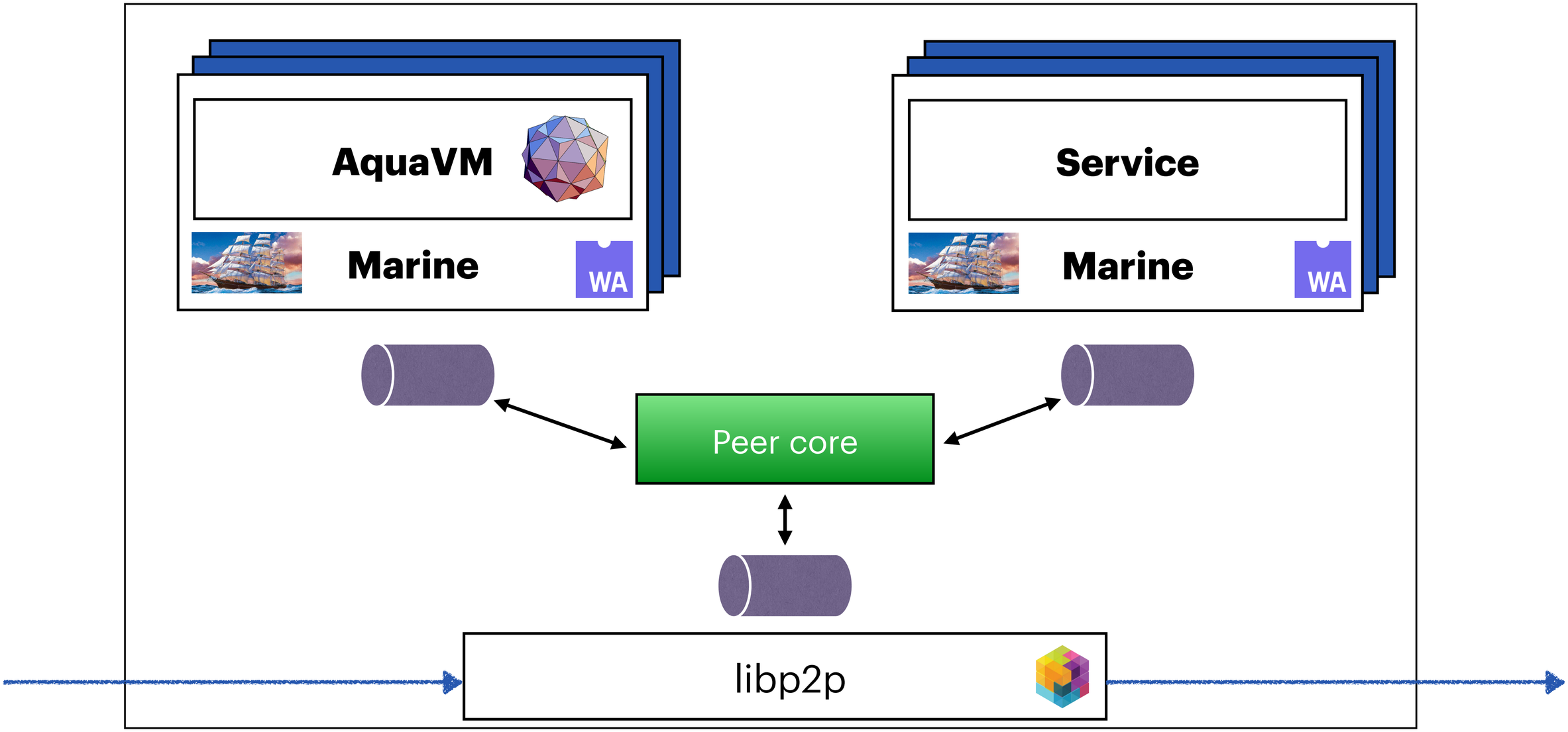
$stream/%map

- CRDT-like (locally-consistent)
- versioned
- could be canonicalized
- without canonicalization could be used only by a few instructions

Compiler theory

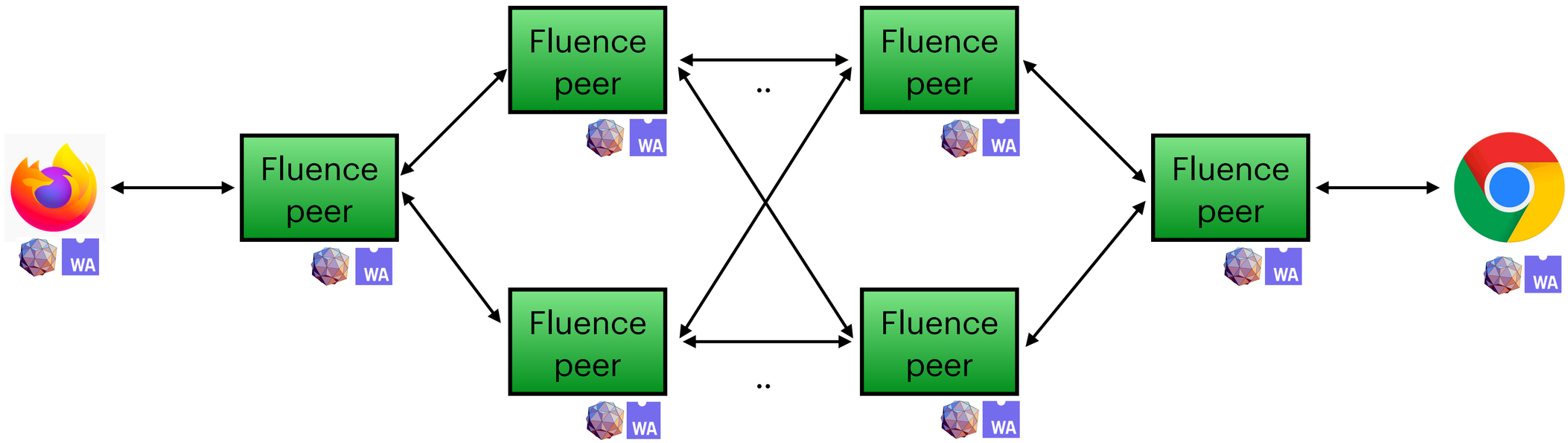
Fluence Labs implementation

Fluence Peer architecture



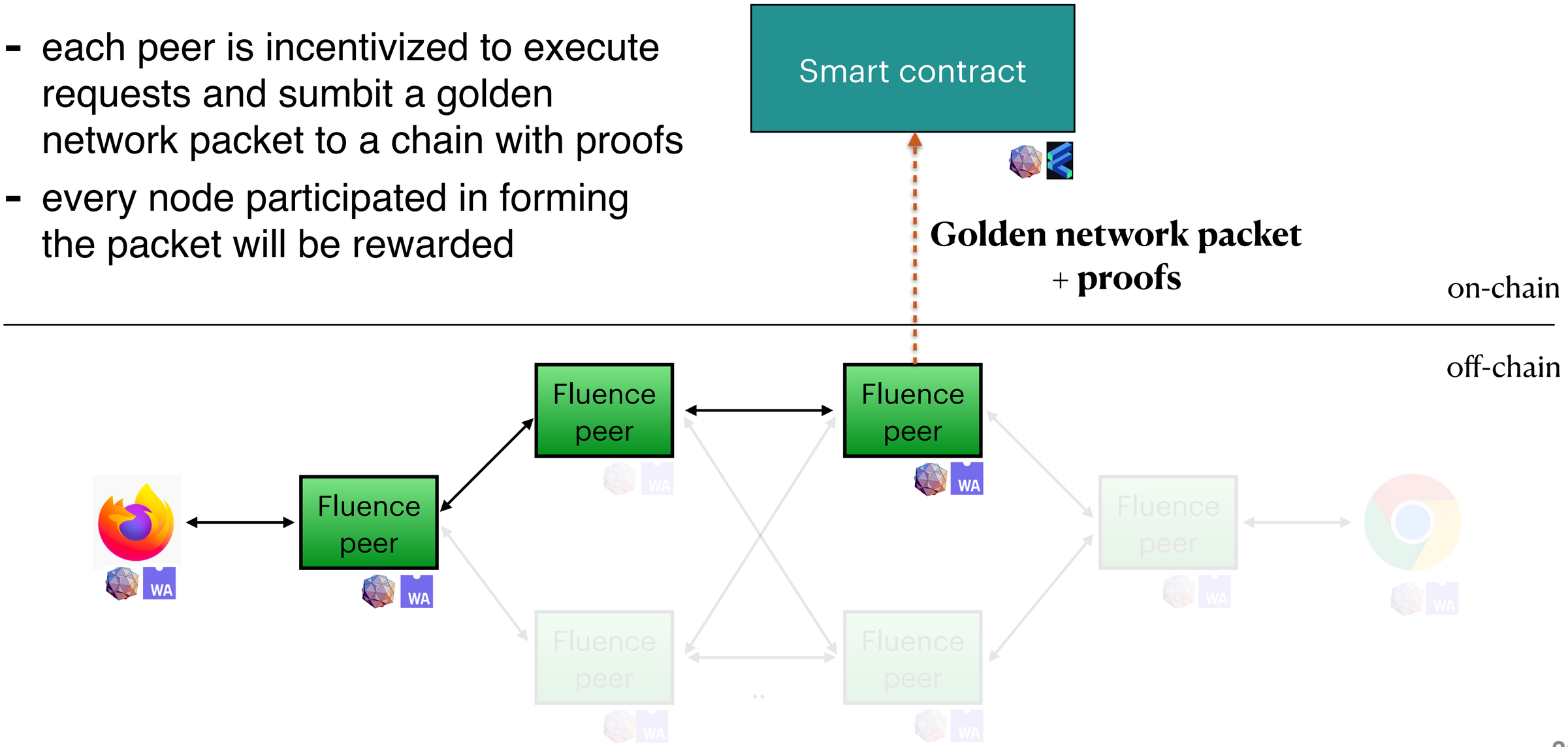
Fluence network

- permisionless
- incentivized
- auditable
- every node is a coordinator
- no implicit consensus



Fluence network

- each peer is incentivized to execute requests and sumbit a golden network packet to a chain with proofs
- every node participated in forming the packet will be rewarded



DCC working group

DCC WG

Distributed Choreography and Composition Working Group

- **Why**
  - DCC is an important component to make distributed and decentralized compute work and enable the use and reuse of (shareable compute) resources
  - Provable and independently verifiable DCC implementations are critical to the broad success of decentralized networks
- **What**
  - Develop and formalize implementations, best practices and (shareable) processes to enable secure, transparent, provable and verifiable workflow engines, processes and patterns independent of the underlying calculi.
  - Tackle challenges such as
    - orchestration models for complex multi-agent permissioned and permissionless p2p networks
    - efficient verification and validation models and processes for both on- and off-chain scenarios
    - formalization approaches over different DCC implementation models for shared reasoning, e.g., via (vanilla) Petri nets or DAGs
    - best practices for payload security and privacy, e.g., e2e

DCC WG: Who Should Care?

Distributed Choreography and Composition Working Group

All Web3 and Web3 stakeholders interested in defining and formalizing a critical part of the decentralized tool and infrastructure stack are welcome and encouraged to participate:

- distributed compute developers

- distributed algorithms developers

- who cares about decentralization of compute

DCC WG

t.me/dcc_wg

github.com/fluencelabs/dcc-wg

my email: mike@fluence.one, tg: @voronovm