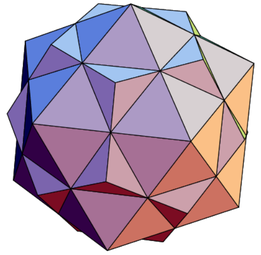
AquaVM: calculus based VM for distributed algorithms

Mike Voronov
twitter.com/@vms11

IPFS þing, July 14, 2022



fluence

Agenda

- Fluence peer architecture
- AquaVM execution model
- AIR instruction
- AquaVM "main principles"

Fluence architecture

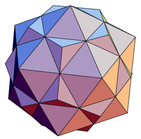
Fluence language stack

code.aqua 

compiled-code.air + ts

execution result

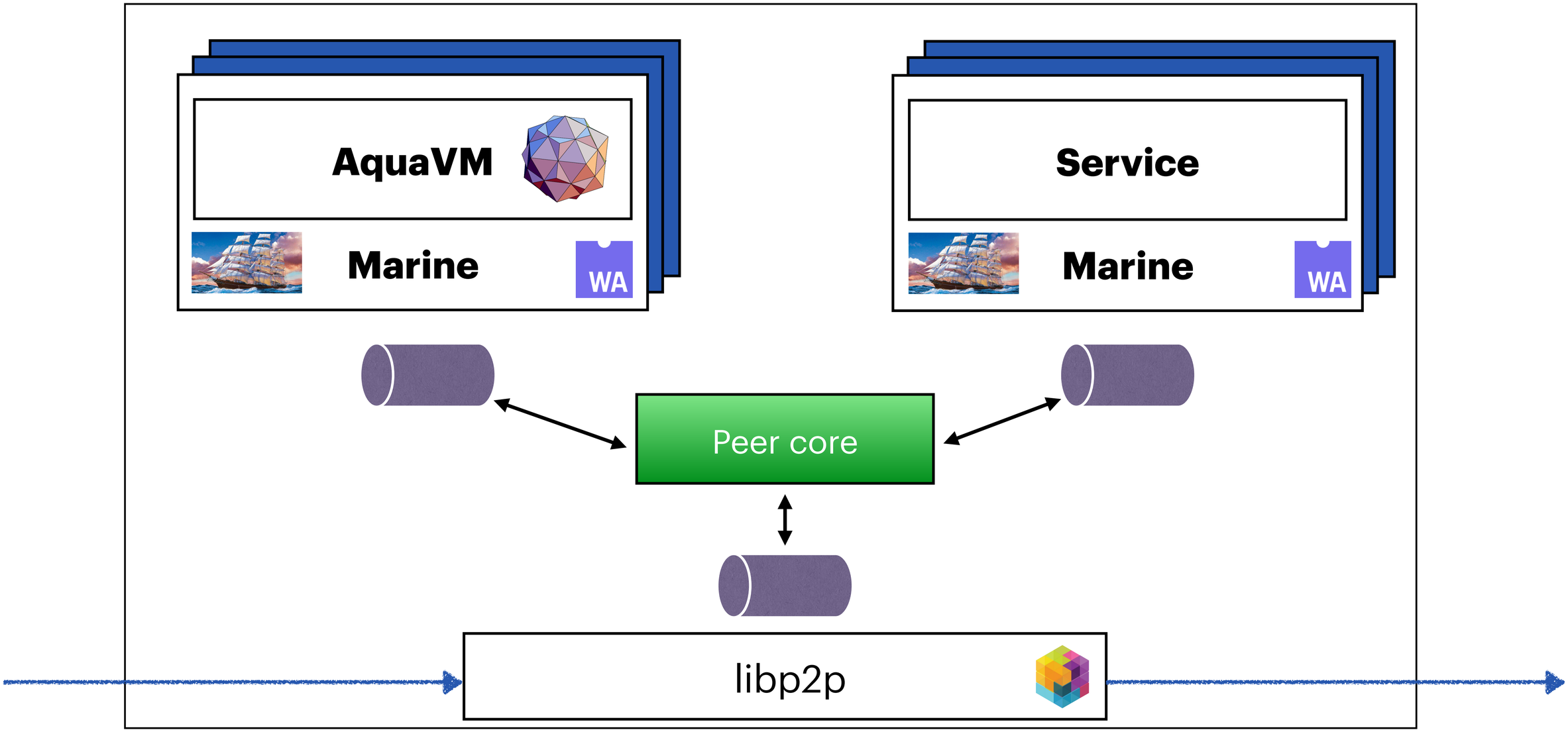
Aqua compiler 



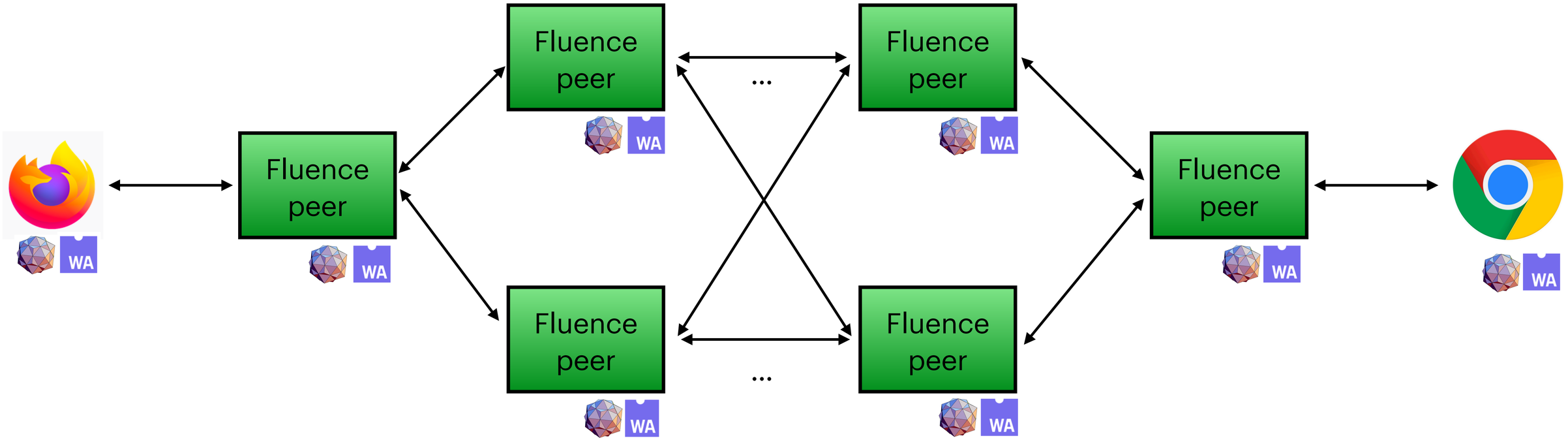
AquaVM 

Marine 

Fluence Peer architecture



Fluence network

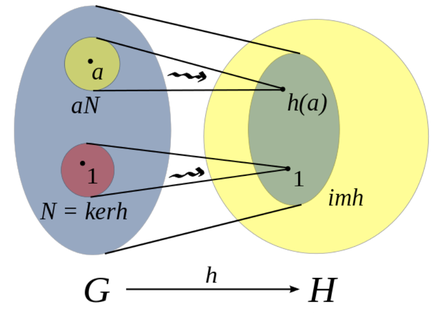


AquaVM's way to rule the complexity

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

f: X -> Y

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 execution model

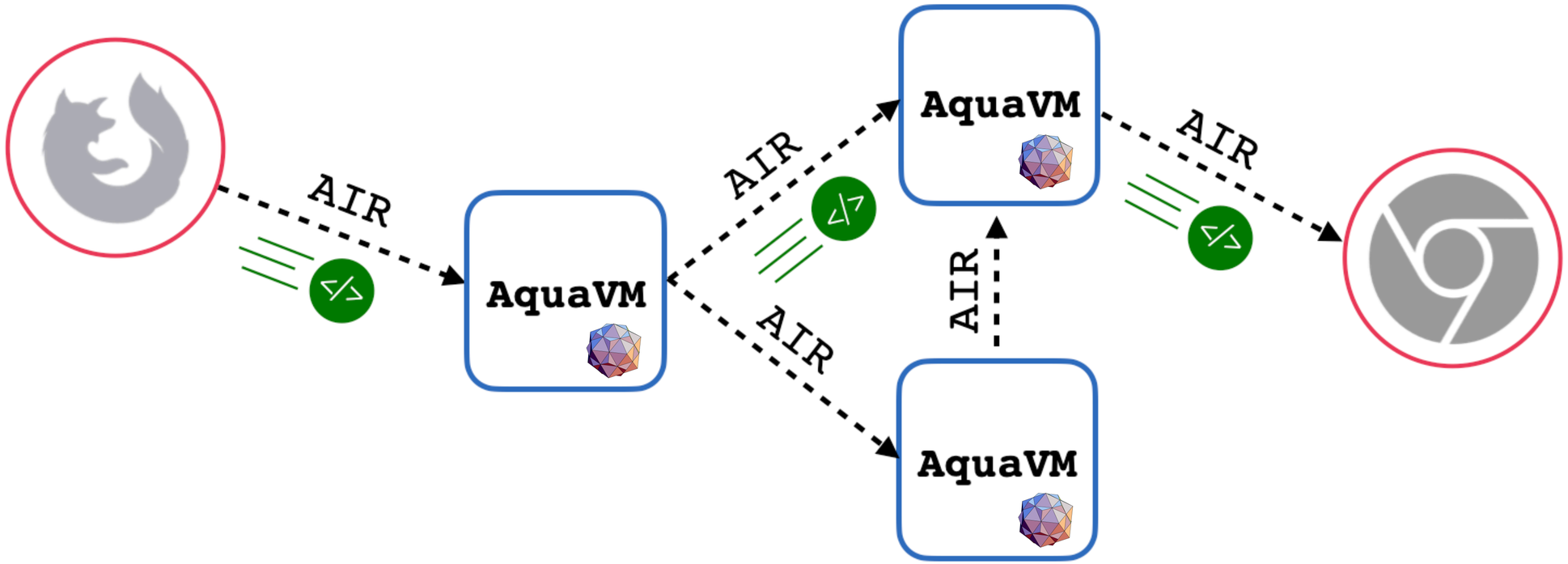
Execution model: particle

- At the core of the distributed programming model is what we call a ***particle***
- Particles are data structures combining **script**, **data**, **init_peer_id**, and some metadata
- Each fields of a particle are immutable except **data**

```
pub struct Particle {
    pub id: String,
    pub timestamp: u64,
    pub ttl: u32,
    pub signature: Vec<u8>,
    pub script: String,
    pub init_peer_id: PeerId,
    pub data: Vec<u8>,
}
```

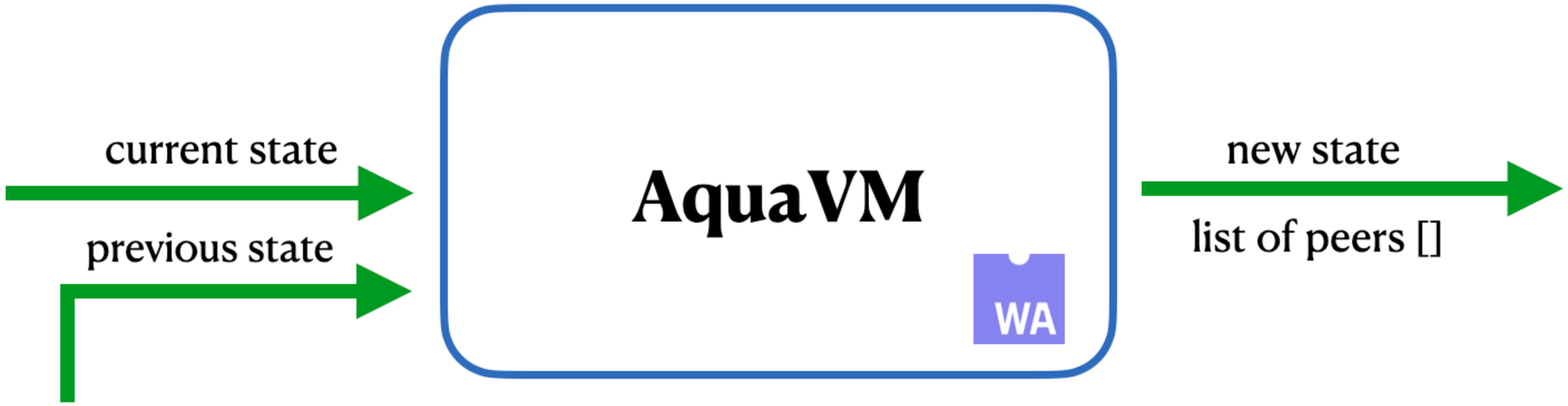
Execution model: particle

Network packets belongs to one particle travel from peer to peer as programmatically specified and resolve along the way, peer by peer



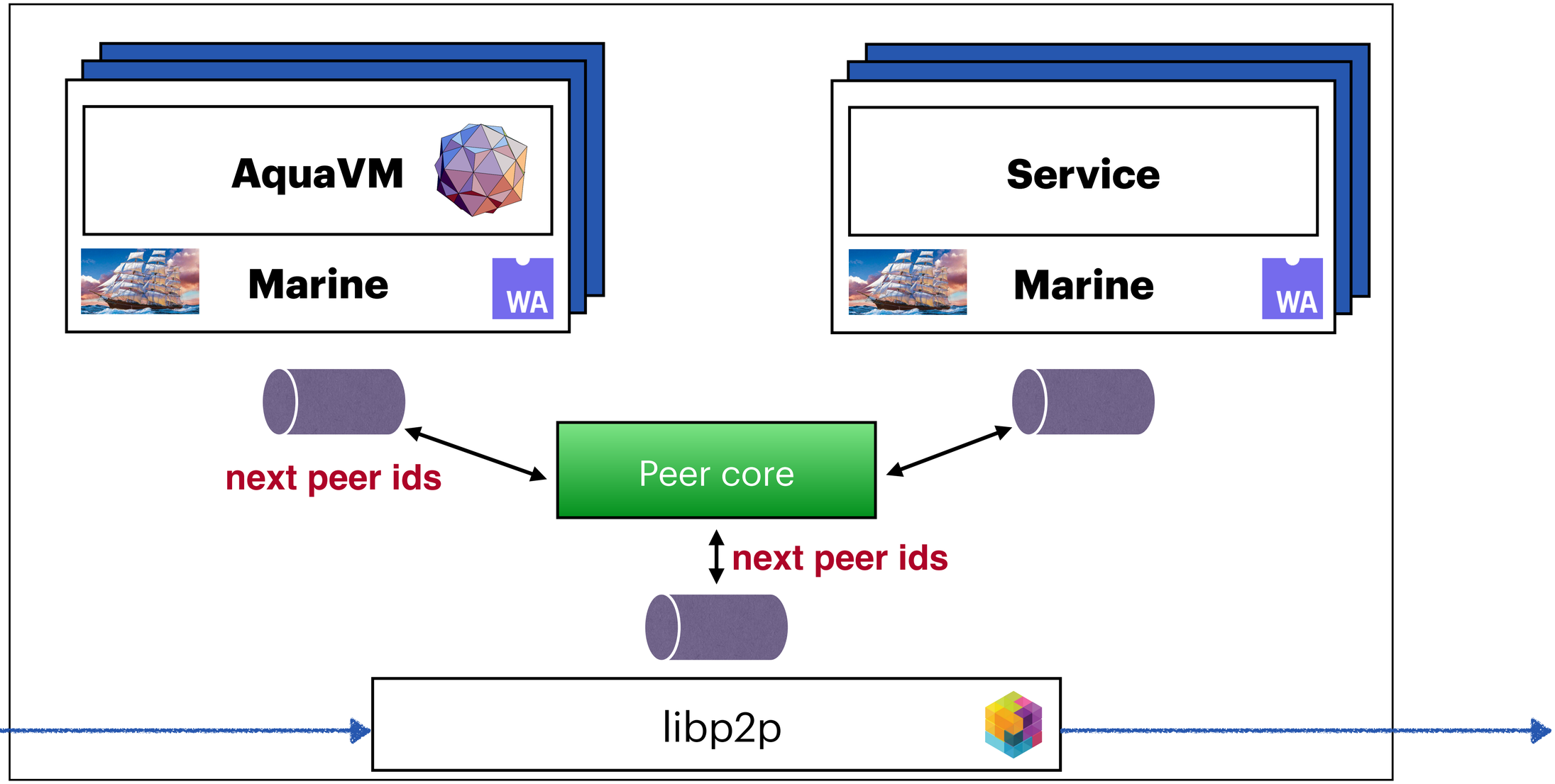
AquaVM as FSM

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



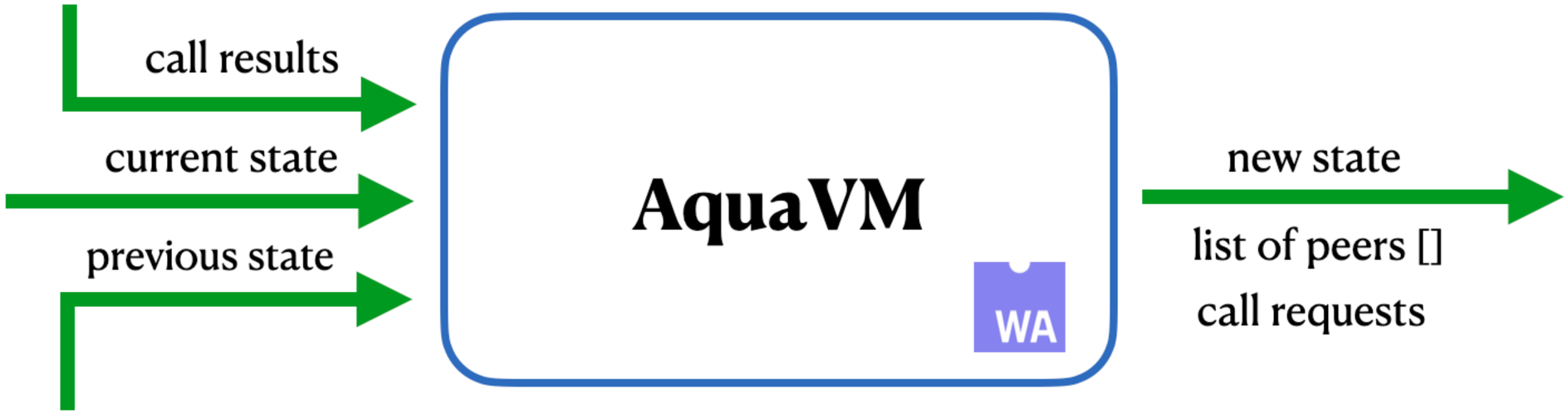
automata theory

Fluence Peer architecture



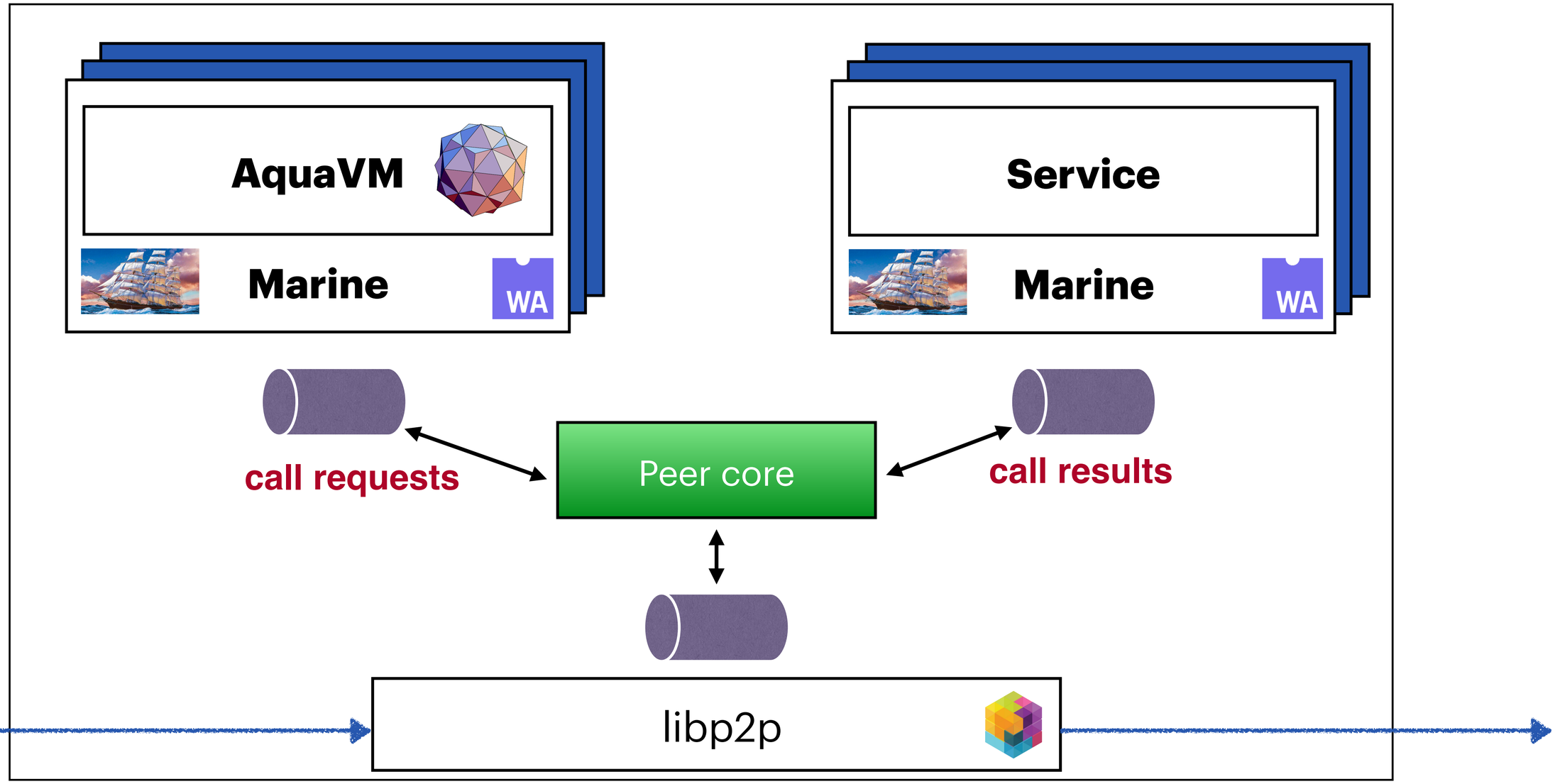
AquaVM as FSM

- 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

Fluence Peer architecture



AquaVM: main principles

There are two reasons to run AquaVM:

- handle data from a new particle
- 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

AIR instructions

pi-calculus: operations

- 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

AIR instructions

- AIR is comprised of a few instructions:
  - **call** - execution
  - **seq** - sequential
  - **par** - parallel
  - **fold** - iteration (for-like)
  - **xor** - catching errors
  - **match/mismatch** - branching
  - **null** - empty instruction (identity)
  - **ap** - applying
  - c10n

pi calculus

AIR instructions: **call**

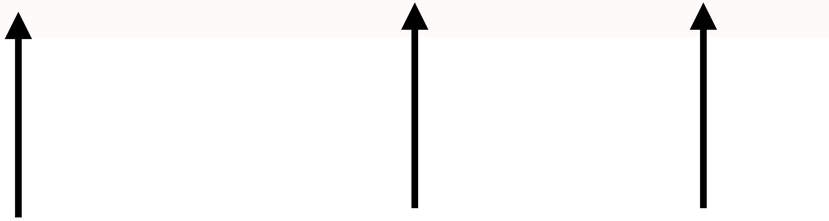
AIR text representation is based on S-expressions (inspired by WAT)

```
(call "peer_id" ("dht" "put") [key value] result)
```

pi calculus

AIR instructions: **call**

```
(call "peer_id" ("dht" "put") [key value] result)
```



location:
peer id

service

function

pi calculus

AIR instructions: **call**

```
(call "peer_id" ("dht" "put") [key value] result)
```

pi calculus

AIR instructions: **call**

```
(call "peer_id" ("dht" "put") [key value] result)
```

location: peer id

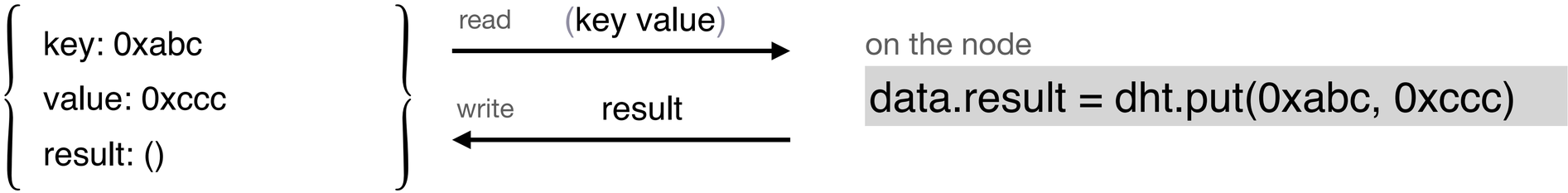
service

function

arguments

output

Execution data



pi calculus

AIR instructions: **seq**

```
(seq
  (call "12D3Node"    ("dht"    "get") [key] value)
  (call "12D3Storage" ("sqlite" "put") [key value])
)
```

- **seq** takes two instructions
- executes them sequentially

pi calculus

AIR instructions: **par**

```
(par
  (call "ClientA" ("chat" "display") [msg])
  (call "ClientB" ("chat" "display") [msg])
)
```

- **par** takes two instructions
- executes them in parallel

pi calculus

AIR instructions: **lambdas**

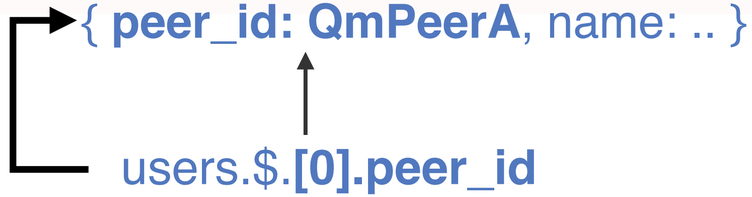
```
(seq
    (call node ("user-list" "get_users") [] users)
    (call users.$.[0].peer_id ("chat" "display") [msg])
)
```

```
users: [
  { peer_id: QmPeerA, name: .. },
  { peer_id: QmPeerB, name: .. },
  ...
]
```

lambda calculus

AIR instructions: **lambdas**

```
(seq
    (call node ("user-list" "get_users") [] users)
    (call users.$.[0].peer_id ("chat" "display") [msg])
)
```




```
users: [
  { peer_id: QmPeerA, name: .. },
  { peer_id: QmPeerB, name: .. },
  ...
]
```

lambda calculus

AIR instructions: **ap**

```
(seq
    (call node ("user-list" "get_users") [] users)
    (ap users.$.[0].peer_id user_0)
)
```

ap applies lambda to the first variable and saves the result in the second

category theory

lambda calculus

AIR instructions: **xor**

```
(xor
    (seq
        (call node ("user-list" "get_users") [] users)
        (ap users.$.[0].peer_id user_0)
    )
    (call %init_peer_id% ("logger" "log") [%last_error%])
)
```

- **xor** takes two instructions
- iff the first instruction fails, the second is executed

pi calculus

AIR instructions: **match/mismatch**

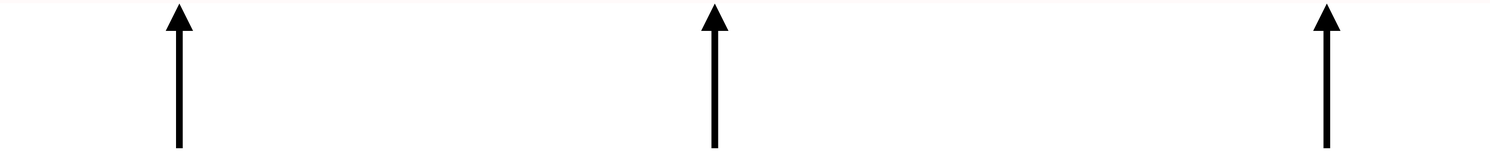
```
(seq
    (call node ("user-list" "get_users") [] users)
    (mismatch users.$.length 0
        (ap users.$.[0].peer_id user_0)
    )
)
```

- **match/mismatch** takes two values and an instruction
- iff they are equals/nonequals executes the provided instruction

pi calculus

AIR instructions: **fold**

```
(fold iterable iterator instruction)
```



array to iterate on

iterator variable

instruction to execute

- **fold** iterates through the array, assigning each element to the iterator
- on each iteration instruction is executed
- instruction can read the iterator

lambda calculus

pi calculus

AIR instructions: iteration

```
(seq
    (call node ("user-list" "get_users") [] users) ;; (1)
    (fold users.$.users u  ;; (2)
        (seq
            (call u.$.peer_id ("chat" "display") [msg]) ;; (3)
            (next u) ;; (4)
        )
    )
)
```

```
users: [
  { peer_id: QmPeerA, name: .. },
  { peer_id: QmPeerB, name: .. },
  ...
]
```

1. Gather chat members by calling user-list.get_users

lambda calculus

pi calculus

AIR instructions: iteration

```
(seq
    (call node ("user-list" "get_users") [] users) ;; (1)
    (fold users.$.users u  ;; (2)
        (seq
            (call u.$.peer_id ("chat" "display") [msg]) ;; (3)
            (next u) ;; (4)
        )
    )
)
```

```
users: [
  { peer_id: QmPeerA, name: .. },
  { peer_id: QmPeerB, name: .. },
  ...
]
```

1. Gather chat members by calling user-list.members
2. Iterate through elements in users array, u = element (json object)

lambda calculus

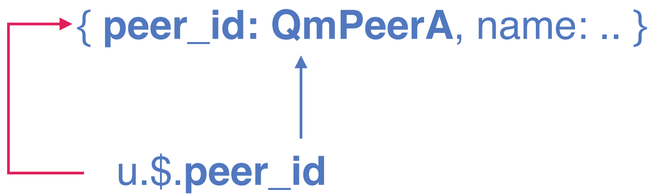
pi calculus

AIR instructions: iteration

```
(seq
    (call node ("user-list" "get_users") [] users) ;; (1)
    (fold users.$.users u  ;; (2)
        (seq
            (call u.$.peer_id ("chat" "display") [msg]) ;; (3)
            (next u) ;; (4)
        )
    )
)
```

```
users: [
  { peer_id: QmPeerA, name: .. },
  { peer_id: QmPeerB, name: .. },
  ...
]
```

1. Gather chat members by calling user-list.members
2. Iterate through elements in users array, u = element
3. Each u is an object; u.$.peer_id reads its field
4. (next u) triggers next iteration



lambda calculus

pi calculus

AIR instructions: **null**

```
(seq
  (null)
  (call "ClientB" ("chat" "display") [msg])
)
```

- null takes no arguments
- does nothing, useful for code generation

AquaVM main principles

AquaVM: value types

Value

scalar

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

```
(seq
  (call %init_peer_id% ("" "load") ["relayId"] relayId)
  (seq
    (call %init_peer_id% ("" "load") ["knownPeers"] knownPeers)
    (seq
      (call %init_peer_id% ("" "load") ["clientId"] clientId)
      ; get info from relay
      (par
```

$stream

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

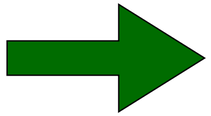
```
(fold $neighs_inner ns
      (seq
       (fold ns n
             (seq
              (call n ("op" "identify") [] $services)
              (next n)
             )
       )
       (next ns)
      )
)
```

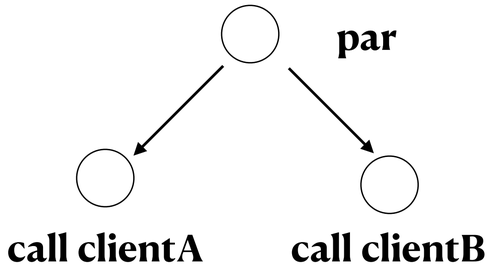
Compiler theory

AquaVM: main principles

Treats instructions as a rooted directed graph

```
(par
  (call "ClientA" ("chat" "display") [msg])
  (call "ClientB" ("chat" "display") [msg])
)
```



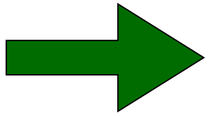


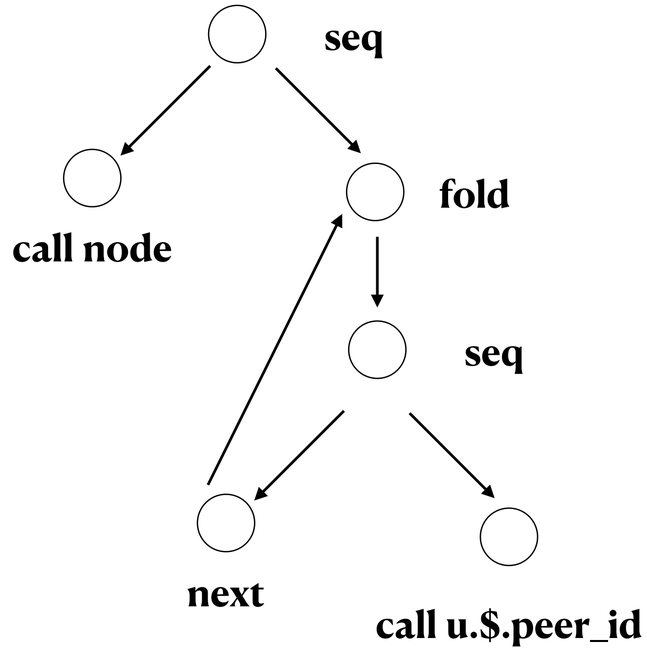
graph theory

AquaVM: main principles

Treats instructions as a rooted directed graph

```
(seq
    (call node ("user-list" "get_users") [] users)
    (fold users.$.users u
        (seq
            (call u.$.peer_id ("chat" "display") [msg])
            (next u)
        )
    )
)
```





graph theory

AquaVM: main principles

- On the execution stage AquaVM tries to cover all subgraphs and execute all calls
- But execution rules are different for different instruction:
  - right subgraph of **seq** will be executed iff left subgraph completed (all calls executed)
  - right subgraph of **xor** will be executed iff left subgraph failed with error
  - **par** always executes both subgraphs

graph theory

AquaVM: main principles

- there is no any saved entry point, each time execution starts from the very beginning of a script
- to not to execute same **call**s twice, execution trace is linearialized and serialized in the following form:

```
pub enum ExecutedState {
    Par(ParResult),
    Call(CallResult),
    Fold(FoldResult),
    Ap(ApResult),
}
```

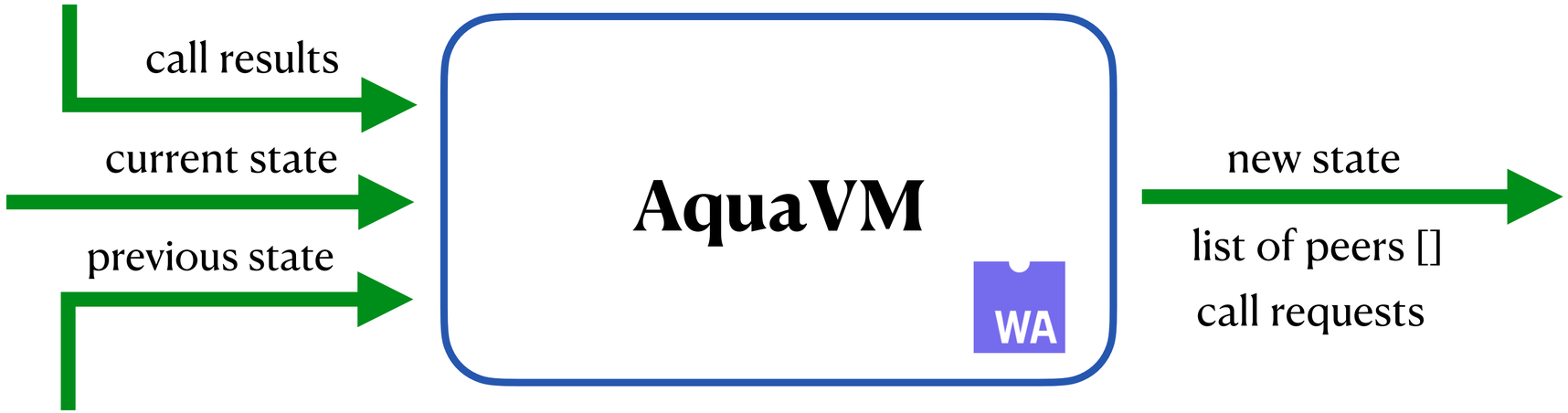
```
pub struct ParResult { pub left_size: u32, pub right_size: u32 }
```

```
pub enum CallResult {
    /// Request was sent to a target node by node with such public key and it shouldn't be called again.
    RequestSentBy(Sender),

    /// A corresponding call's been already executed with such value as a result.
    Executed(Value),

    /// call_service ended with a service error.
    CallServiceFailed(i32, Rc<String>),
}
```

graph theory

AquaVM: main principles

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

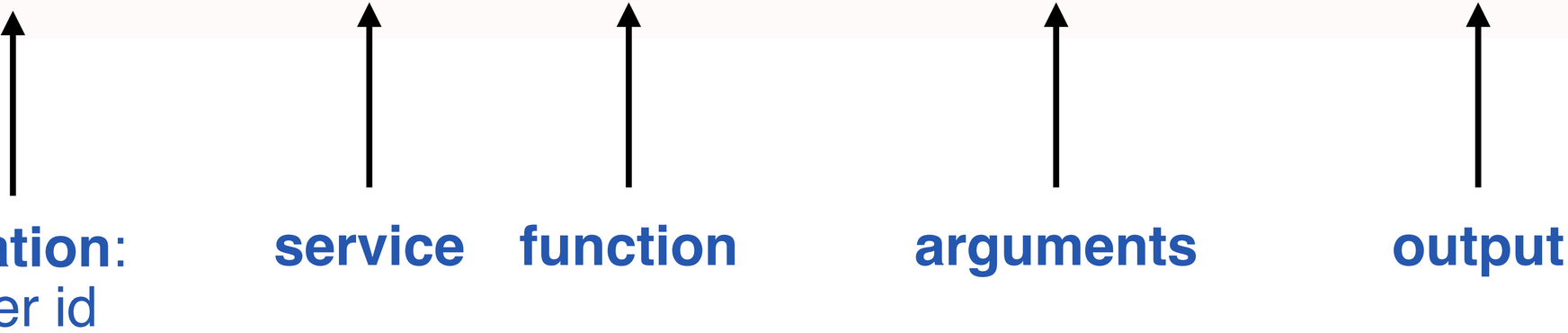


automata theory

AquaVM: main principles

- If peer_id is **equal** to current node peer_id it's parameters will be collected
- If peer_id is **not equa**l to current node **peer_id** will be added to **next_peer_ids**

```
(call "peer_id" ("dht" "put") [key value] result)
```



AquaVM: main principles

There are two reasons to run AquaVM:

- handle data from a new network packet
- 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

automata theory

AquaVM: main principles

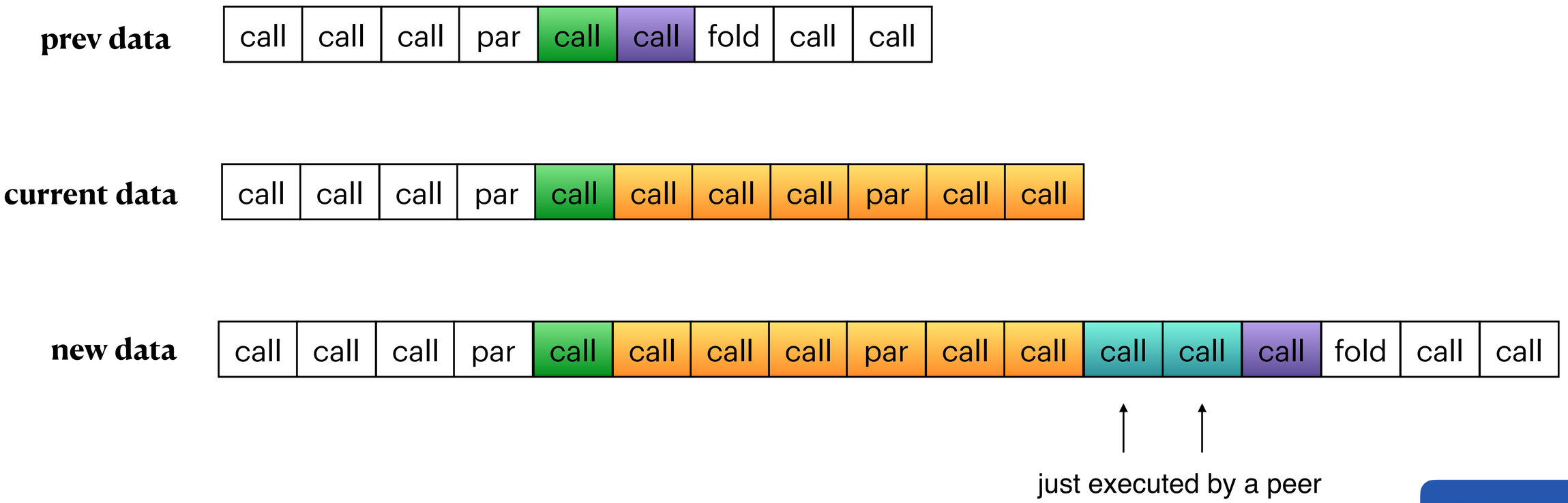
- during the execution of a script AquaVM also **merges** current and previous data provided
- the resulting data should be saved by a peer and passed back to AquaVM as a previous data

```rust
fn execute_aqua_impl(
    init_peer_id: String,
    aqua: String,
    prev_data: Vec<u8>,
    data: Vec<u8>,
) -> Result<InterpreterOutcome, InterpreterOutcome> {
    let PreparationDescriptor {
        mut exec_ctx : ExecutionCtx ,
        mut trace_ctx : ExecutionTraceCtx ,
        aqua : Instruction ,
    } = prepare(&prev_data, &data, aqua.as_str(), init_peer_id)
        // return the initial data in case of errors
        .map_err(|e : PreparationError | outcome::from_preparation_error(data, e))?;

    aqua.execute(&mut exec_ctx, &mut trace_ctx)
        // return new collected trace in case of errors
        .map_err(|e : Rc<ExecutionError> | outcome::from_execution_error(&trace_ctx.

    let outcome : InterpreterOutcome = outcome::from_path_and_peers(&trace_ctx.new_

    Ok(outcome)
}
```

automata theory

AquaVM: main principles

While execution AquaVM merges the previous (saved on a peer) data with current data from incoming particle and add just executed on this peer states



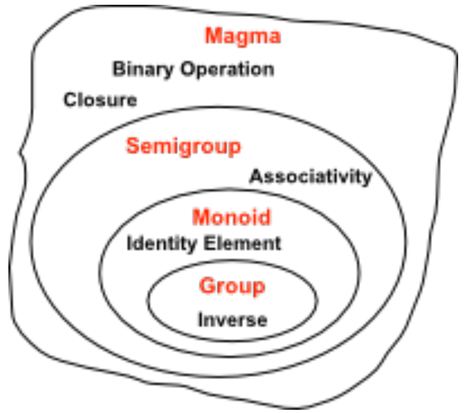
automata theory

AquaVM: "math" point of view

AquaVM could be considered as a function (magma) $f : X * X \to X$

(X, f) is a non-commutative idempotent **monoid** (LBRM):

1. f is **associative**: $\forall a, b, c \in X \; f(f(a, b), c) = f(a, f(b, c))$
2. f has a **neutral** element: $\exists e \in X \forall a \in X : f(e, a) = f(a, e) = a$
3. f is a **non-commutative** function: $\exists a, b \in X : f(a, b) \neq f(b, a)$
4. f is satisfied these **idempotence** properties:
   - $\forall x \in X : f(x, x) = x$
   - $\forall a, b \in X : f(a, b) = c \in X \Rightarrow f(c, b) = c, f(c, a) = c$



abstract algebra

AquaVM: "math" point of view

1. **f** is **closed** on X

guarantees that AquaVM always returns a data even if data is malformed or some error occurred

```
fn execute_air_impl(...) -> Result<InterpreterOutcome, InterpreterOutcome> {
    let PreparationDescriptor {
        mut exec_ctx,
        mut trace_handler,
        air,
    } = match prepare(&prev_data, &data, air.as_str(), &call_results, params) {
        Ok(desc) => desc,
        // return the initial data in case of errors
        Err(error) => return Err(outcome::from_preparation_error(prev_data, error)),
    };

    match air.execute(&mut exec_ctx, &mut trace_handler) {
        Ok(_) => try_make_outcome(exec_ctx, trace_handler),
        // return the old data in case of any trace errors
        Err(e) if !e.is_catchable() => Err(outcome::from_trace_error(prev_data, e)),
        // return new collected trace in case of errors
        Err(e) => Err(outcome::from_execution_error(exec_ctx, trace_handler, e)),
    }
}
```

abstract algebra

AquaVM: "math" point of view

2. **f** is **associative:** $\forall a, b, c \in X \; f(f(a, b), c) = f(a, f(b, c))$

data merging order doesn't matter => fork-join scheme is expressible

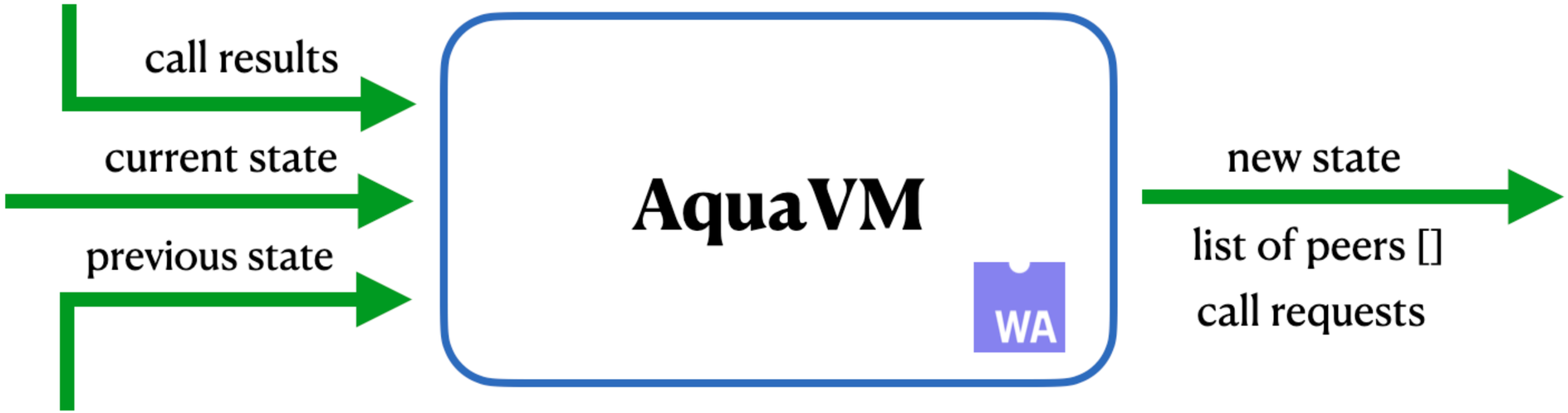
```
(seq
    (seq
        (call node ("user-list" "get_users") [] users)
        (fold users.$.users u
            (seq
                (call u.$.peer_id ("chat" "reply") [msg] $result_stream)
                (next u)
            )
        )
    )
    (call %init_peer_id% ("" "") [])
)
```

abstract algebra

AquaVM: "math" point of view

3. f has a **neutral** element: $\exists e \in X \forall a \in X : f(e,a) = f(a,e) = a$

Once call_results are ready, they can be passed into AquaVM with empty current_data

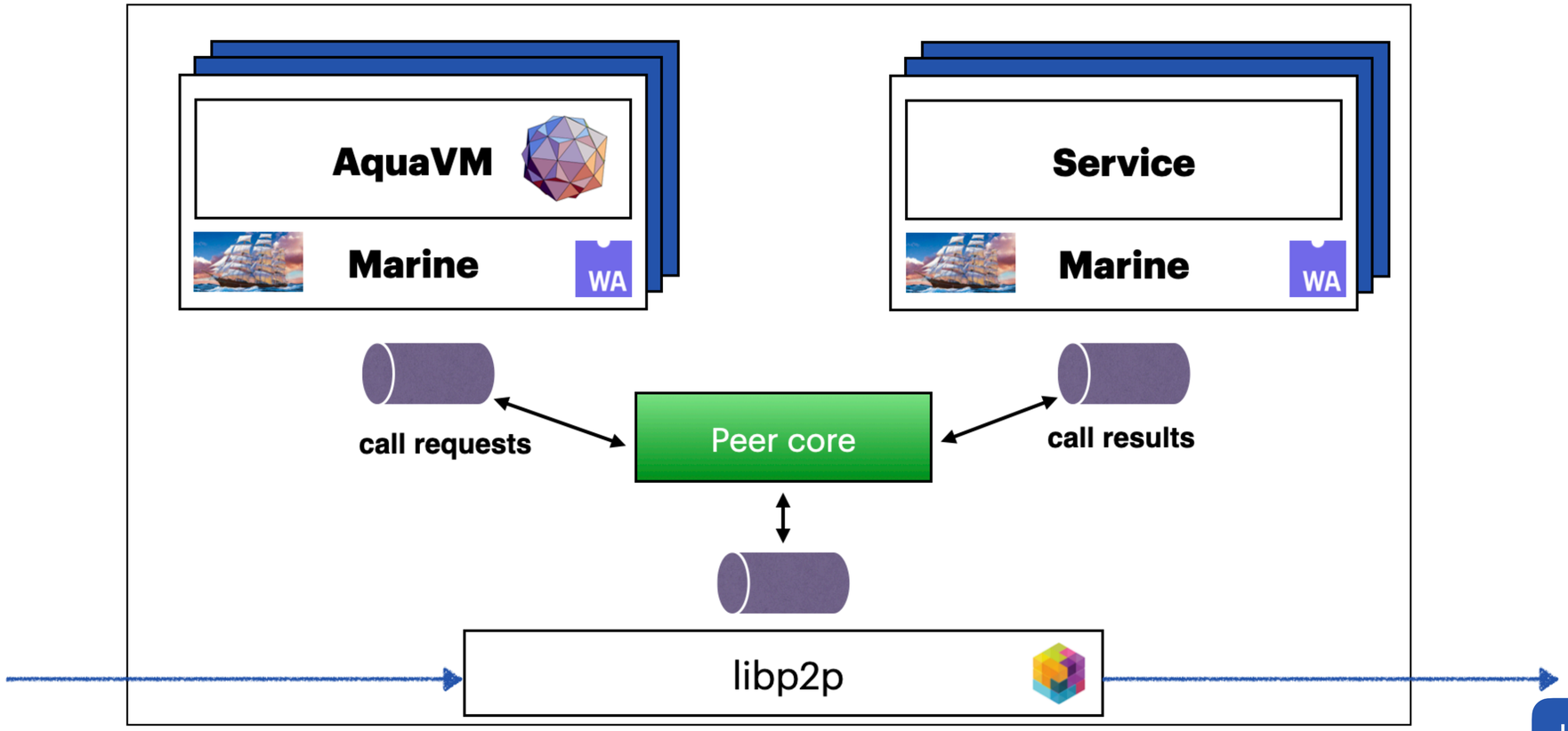


abstract algebra

AquaVM: "math" point of view

3. f has a **neutral** element: $\exists e \in X \forall a \in X : f(e,a) = f(a,e) = a$

Once call_results are ready, they can be passed into AquaVM with empty current_data



abstract algebra

AquaVM: "math" point of view

4. f is a **non-commutative** function: $\exists a, b \in X : f(a, b) \neq f(b, a)$

AquaVM "keeps" all it state in a new data that host should preserve and return back as a previous data

abstract algebra

AquaVM: tetraplets

Each value in AIR has attached tetraplets that describes the origin set this variable

```rust
pub struct SecurityTetraplet {
    /// Id of a peer where corresponding value was set.
    pub peer_pk: String,

    /// Id of a service that set corresponding value.
    pub service_id: String,

    /// Name of a function that returned corresponding value.
    pub function_name: String,

    /// Value was produced by applying this `json_path` to
    /// the output from `call_service`.
    pub json_path: String,
}
```

AquaVM: tetraplets

- A tetraplet is a low-level primitive that could be used to achive fine-grained security model
- Tetraplets are passed to a service during a service call
- A service can determine then whether the requested operation is authorized

```
(seq
    (call "auth_peer_id" ("auth" "is_authorized") [] auth_result)
    (call "log_peer_id" ("log_storage" "delete") [auth_result.$.is_authorized])
)
```

The project overview

AquaVM - github.com/fluencelabs/aquavm

Aqua - github.com/fluencelabs/aqua

The project updates

- twitter.com/fluence_project
- fluence.network/ (newsletter)
- t.me/fluencedev - Fluence developer updates

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

Thanks!

`fluence.network`