COP for Smart Contracts
Activity Contexts

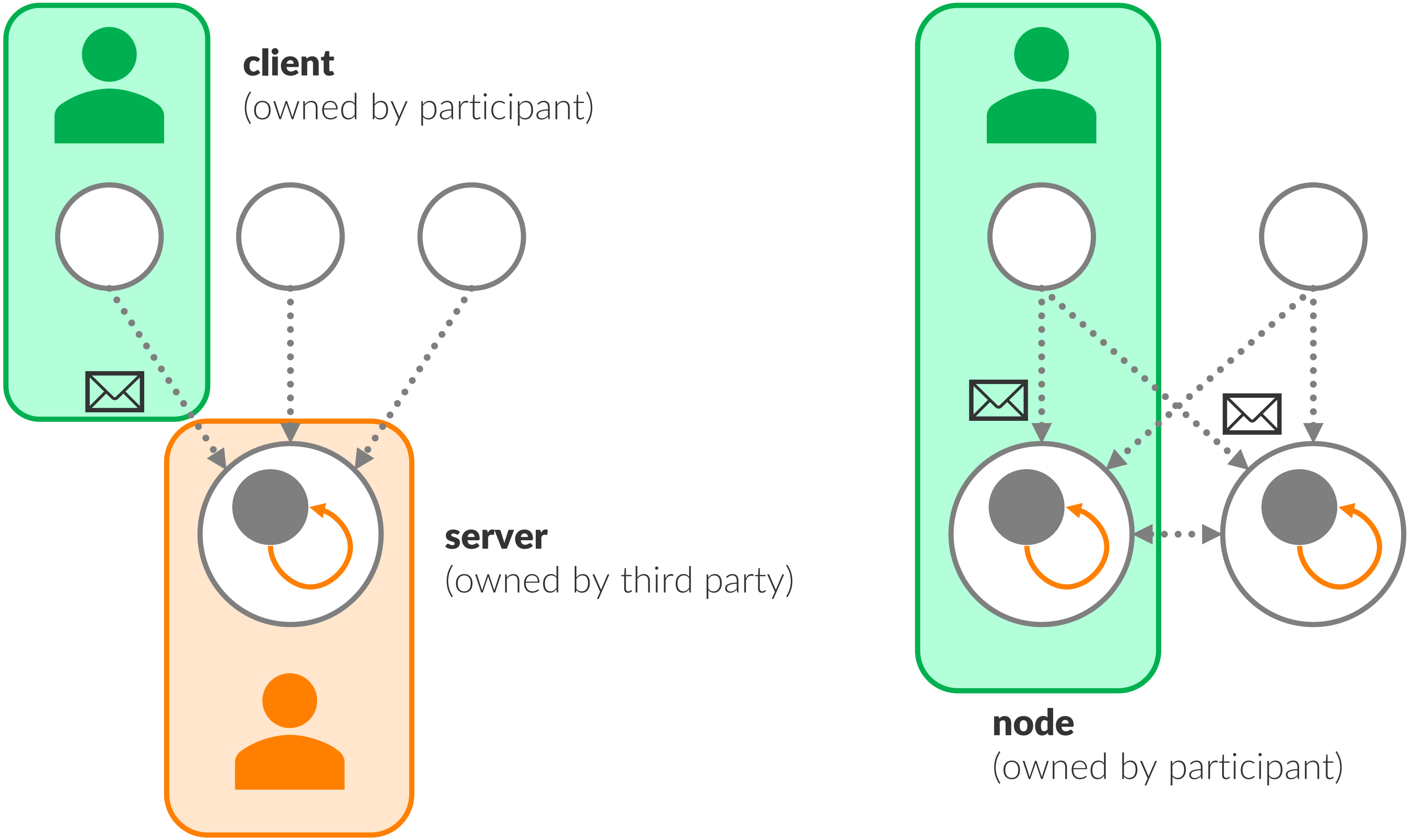
Toni Mattis, Robert Hirschfeld

Software Architecture Group

Hasso Plattner Institute, University of Potsdam, Germany

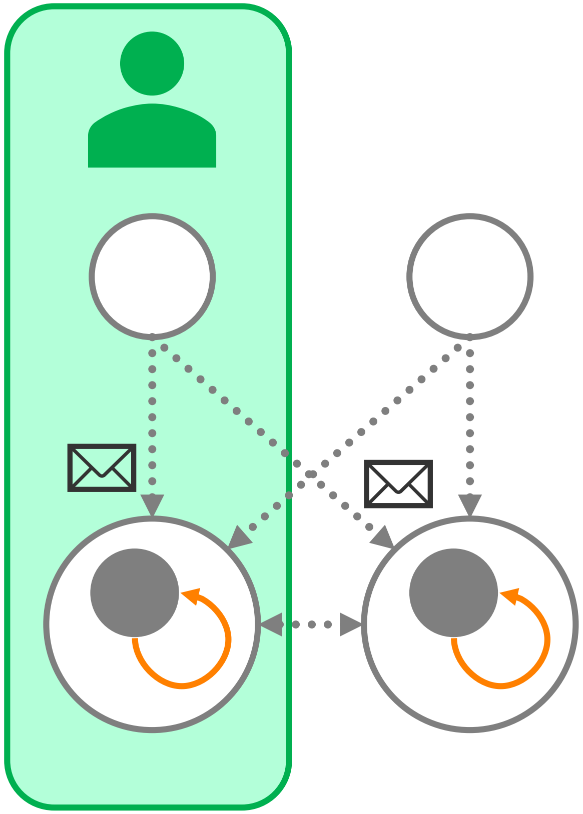
COP '18 17 July 2018, Amsterdam, Netherlands

Centralized vs. Decentralized Services



Smart Contracts as Decentralized Service

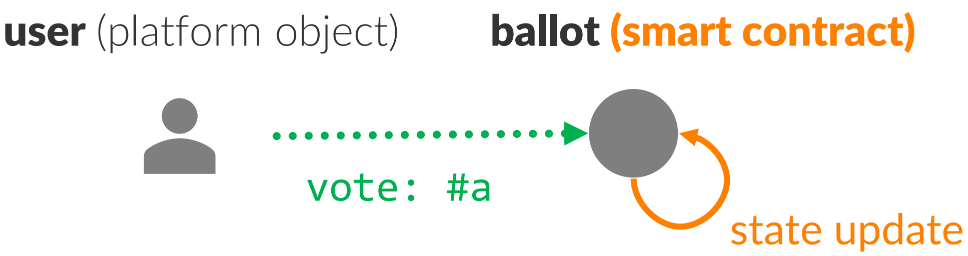
- Set of executable rules according to which real-world actors can interact
  - "Game" (state, moves, players)
  - "Object" (identity, state, behavior)
- Automated enforcement
  - Transfer digitally manageable goods (money, rights, ...)
  - Can take external events as input (deadlines, stock prices, ...)
- No central authority
  - Consensus by quorum
  - Lower transaction costs
  - Trustless



node
(owned by participant)

Decentralized Execution Model

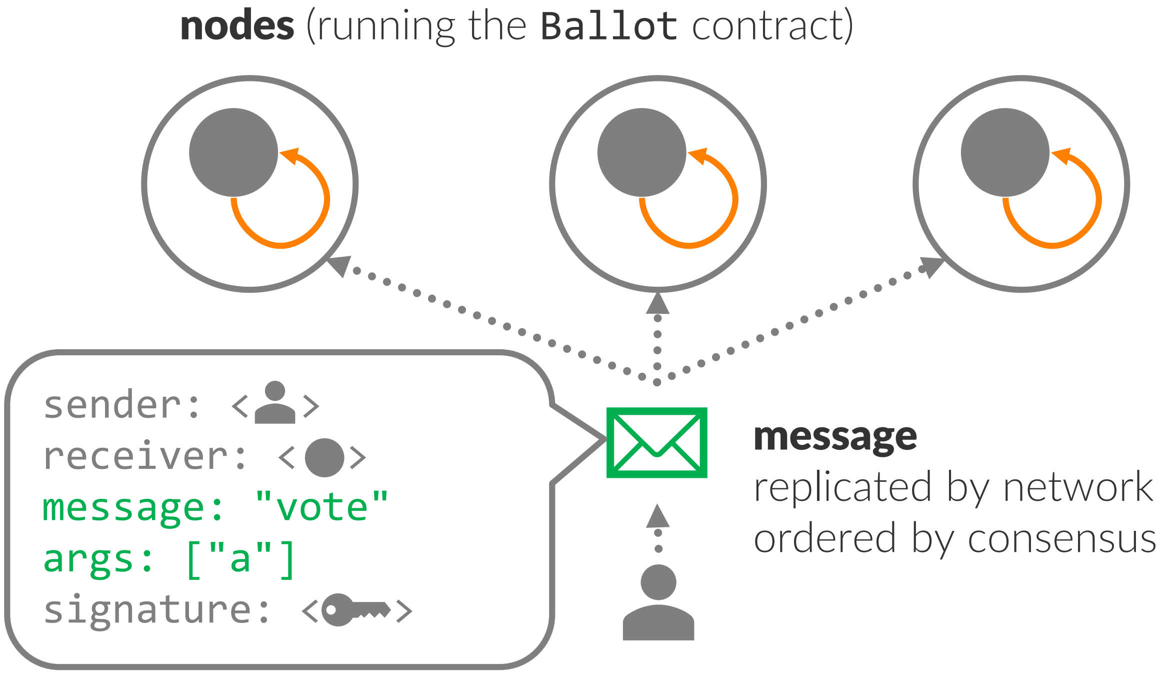
logical perspective
objects and messages



user (platform object)

ballot **(smart contract)**

distribution perspective
replicated copies and messages



nodes (running the `Ballot` contract)

```
sender: <👤>
receiver: <●>
message: "vote"
args: ["a"]
signature: <🔑>
```

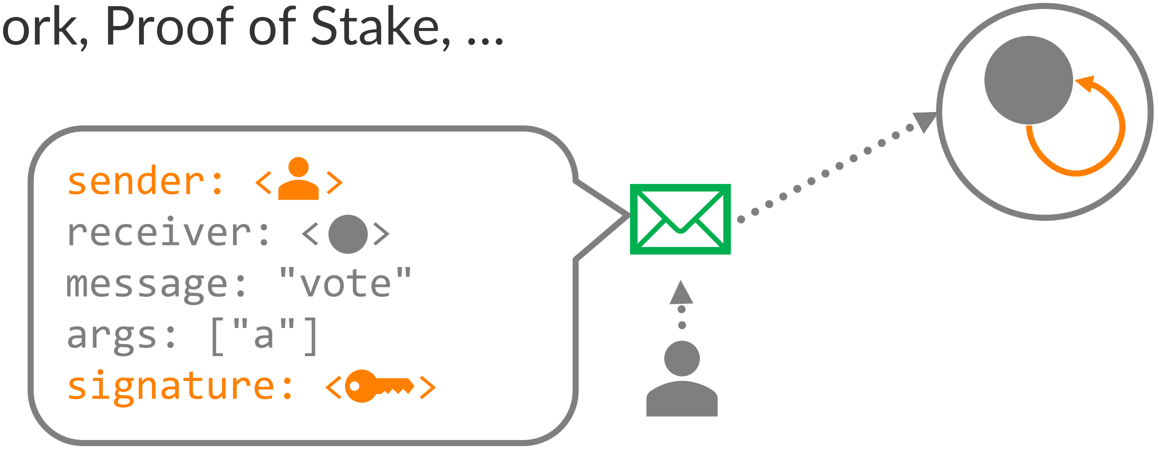
message
replicated by network
ordered by consensus

Security and Consensus

- User identity linked to public key
  - Same public keys = same user
  - User signs all messages using corresponding private key

- Consensus protocol establishes a unique global order of messages
  - Paxos, Byzantine Fault Tolerance (BFT)
  - Proof of Work, Proof of Stake, …

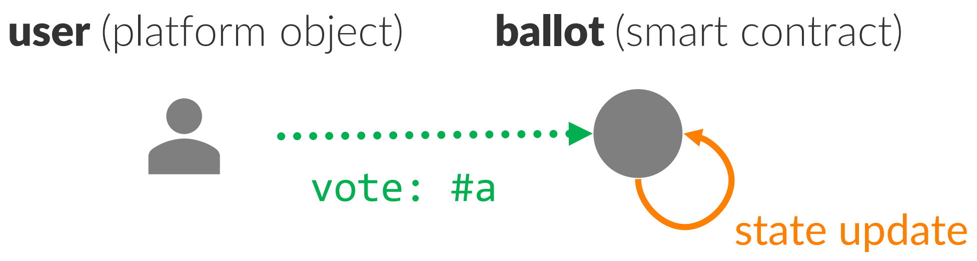
Blockchain



```
sender: <👤>
receiver: <●>
message: "vote"
args: ["a"]
signature: <🔑>
```

Decentralized Execution Model

logical perspective
objects and messages

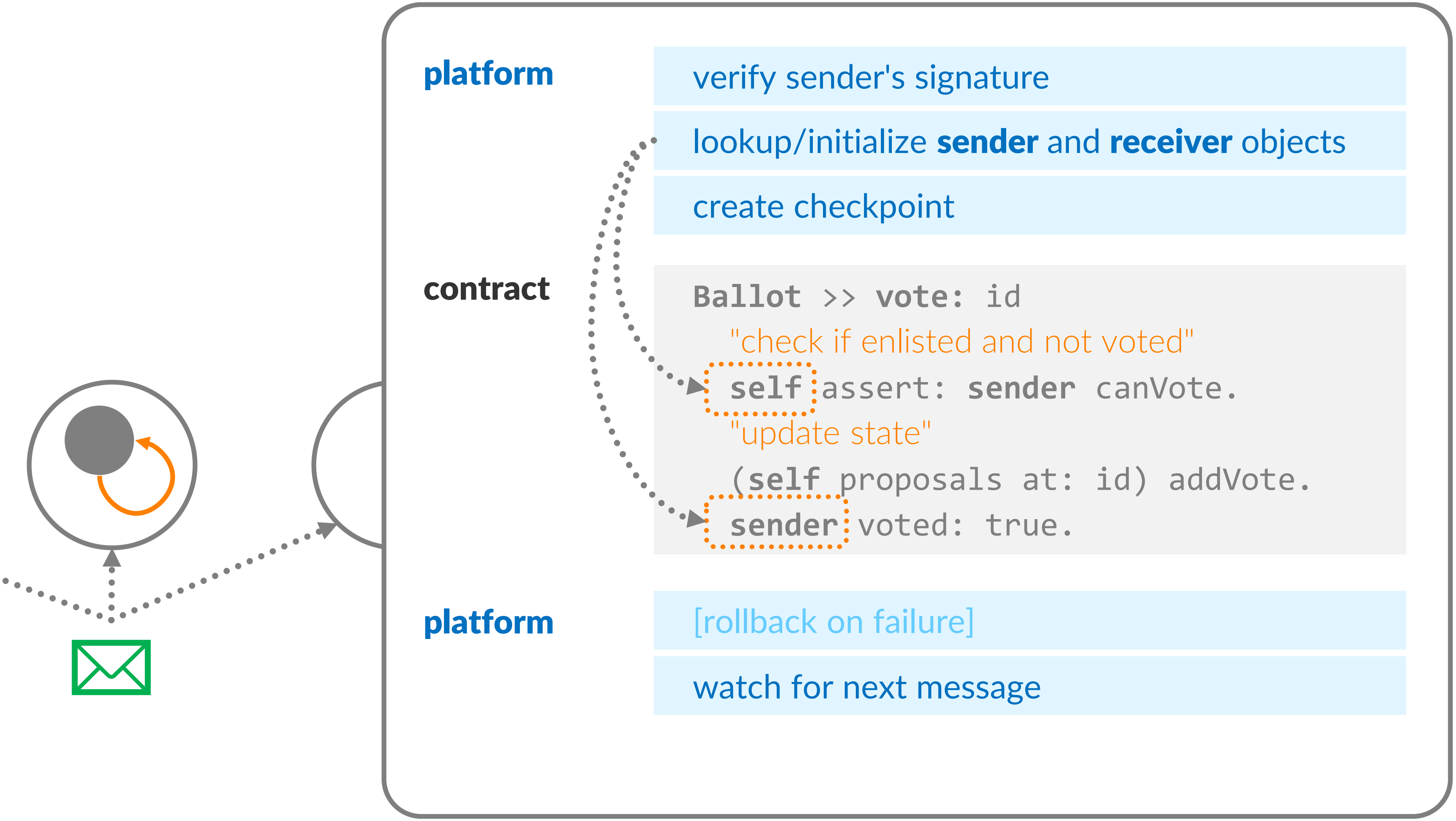


```
Ballot >> vote: id
    "check if enlisted and not voted"
    self assert: ~~sender canVote.~~
    "update state"
    (self proposals at: id) addVote.
    ~~sender voted: true.~~
```

instance of class **User**
provided by platform, not modifiable

How can we add state & behavior?

Decentralized Execution Model



platform

- verify sender's signature
- lookup/initialize **sender** and **receiver** objects
- create checkpoint

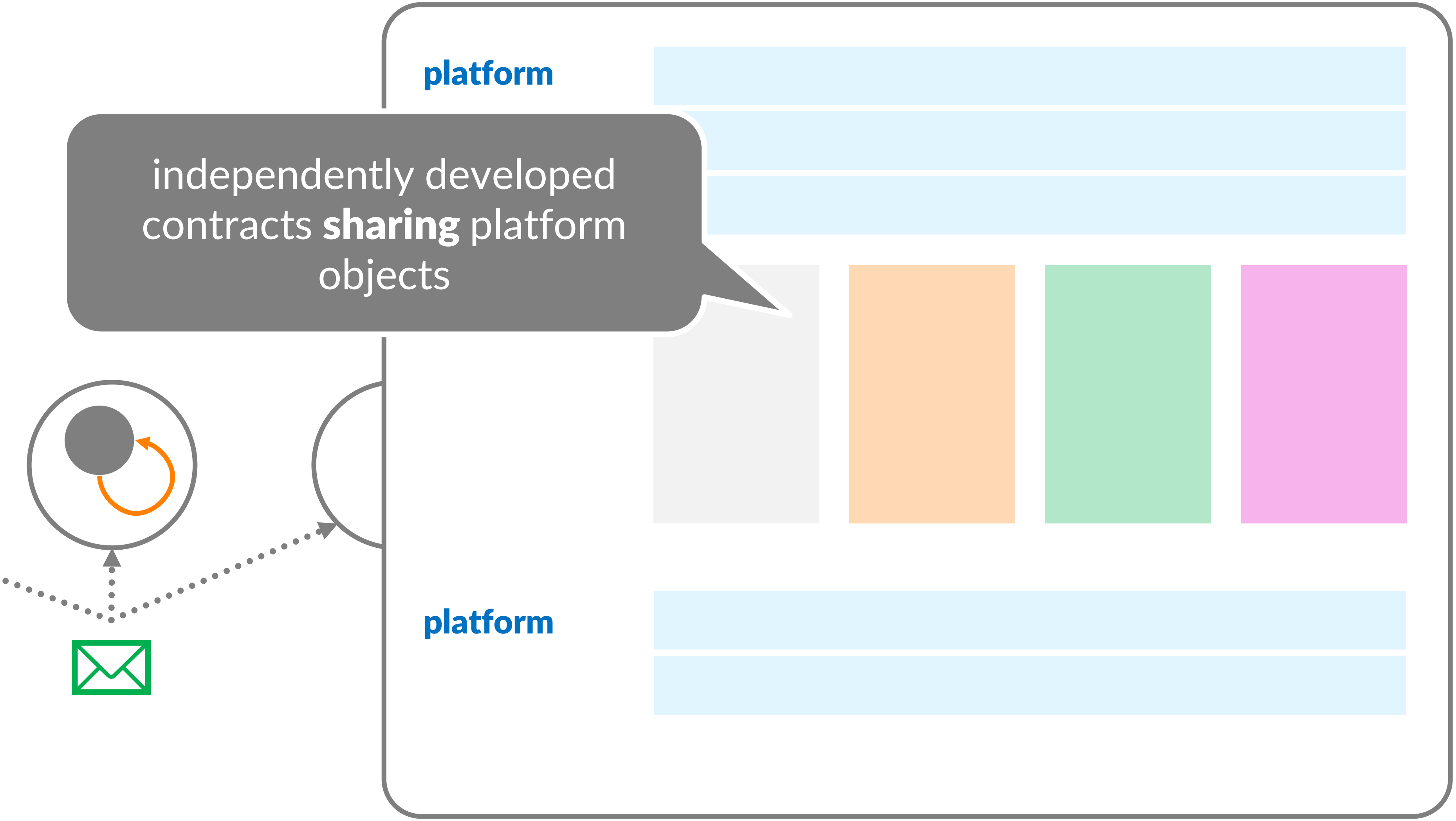
contract

```
Ballot >> vote: id
   "check if enlisted and not voted"
   self assert: sender canVote.
   "update state"
   (self proposals at: id) addVote.
   sender voted: true.
```

platform

- [rollback on failure]
- watch for next message

Decentralized Execution Model



independently developed contracts **sharing** platform objects

Current Workaround: Mediator

Dictionary (voters) with user information

```
Ballot >> vote: id
  | user |
  user := self voters at: sender address.
  "check if enlisted and not voted"
  self assert: user canVote.
  "update state"
  (self proposals at: id) addVote.
  user voted: true.
```

- Lack of encapsulation
- Tendency to drift towards data classes and god-class like mediator

Example in Practice (Solidity)

```
 /// @title Voting with delegation.
contract Ballot {
    // This declares a new complex type which will
    // be used for variables later.
    // It will represent a single voter.
    struct Voter {
        uint weight; // weight is accumulated by delegation
        bool voted;  // if true, that person already voted
        address delegate; // person delegated to
        uint vote;   // index of the voted proposal
    }

    […]

    // This declares a state variable that
    // stores a `Voter` struct for each possible address.
    mapping(address => Voter) public voters;

    […]
```

https://solidity.readthedocs.io/en/v0.4.21/solidity-by-example.html

Decentralized Execution Model

We want to add **behavior** ...

```
User >> canVote
  ^self eligible and:
   [self voted not]
```

and **state** to a platform object
in the context of the voting **activity**

verify sender's signature

lookup/initialize **sender** and **receiver** objects

create checkpoint

```
Ballot >> vote: id
    "check if enlisted and not voted"
    self assert: sender canVote.
    "update state"
    (self proposals at: id) addVote.
    sender voted: true.
```

[rollback on failure]

watch for next message

Activity Contexts

extend `User` objects in the context of `Ballot`
(= during the voting activity)

```
Ballot >> User >> canVote
  ^self eligible and:
   [self voted not]
```

```
Ballot >> vote: id
   "check if enlisted and not voted"
   self assert: sender canVote.
   "update state"
   (self proposals at: id) addVote.
   sender voted: true.
```

behavior and state visible in control flows
originating from `Ballot`

Activity Contexts

```
Ballot >> User >> canVote
  ^self eligible and:
   [self voted not]
```

```
Ballot >> User >> eligible
  <activityAccessor>
  ^false
```

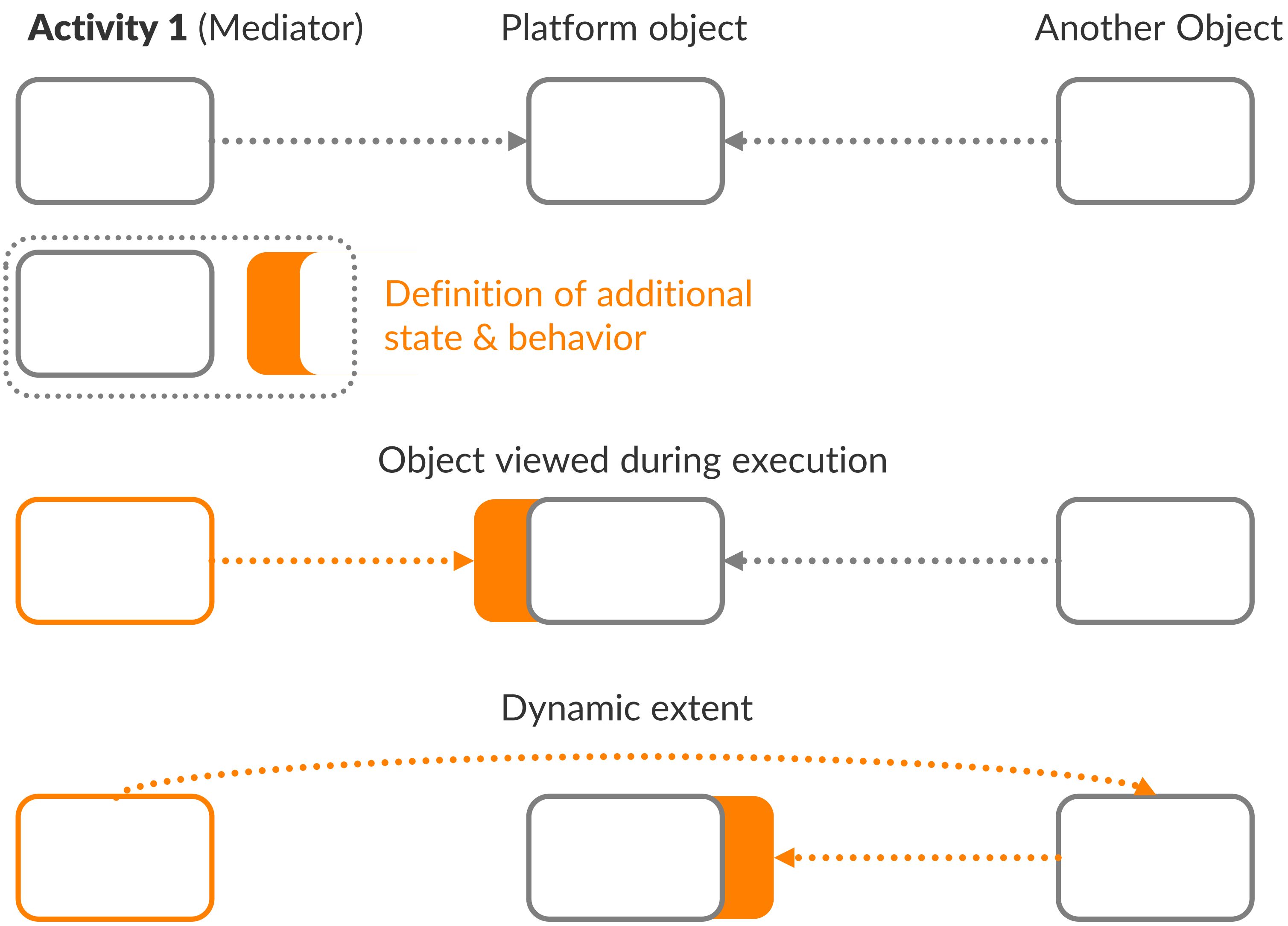
```
Ballot >> User >> voted
  <activityAccessor>
  ^false
```

```
Ballot >> vote: id
  "check if enlisted and not voted"
  self assert: sender canVote.
  "update state"
  (self proposals at: id) addVote.
  sender voted: true.
```

state (accessors)

default value (when the object enters the activity first)

Activity Contexts: Dynamic Extent



Activity 1 (Mediator)

Platform object

Another Object

Definition of additional state & behavior

Object viewed during execution

Dynamic extent

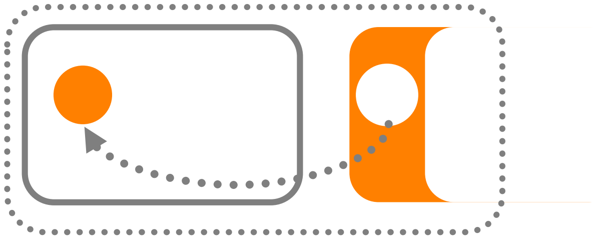
Activity Contexts: State Scoping

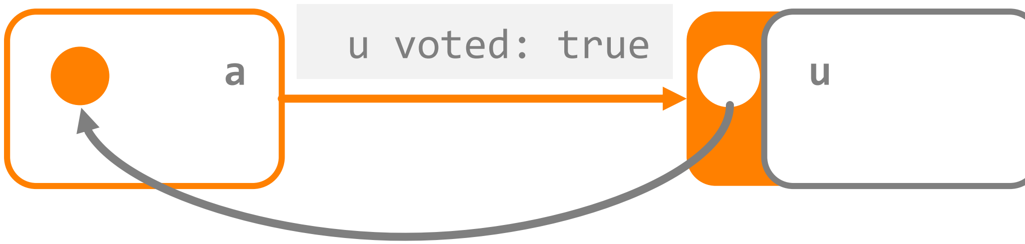
Platform objects may be immutable, where do we store state?

```
Ballot >> User >> eligible
  <activityAccessor>
  ^false
```

```
Ballot >> User >> voted
  <activityAccessor>
  ^false
```

State remains (lexically) scoped to the activity





```
u voted: true
```

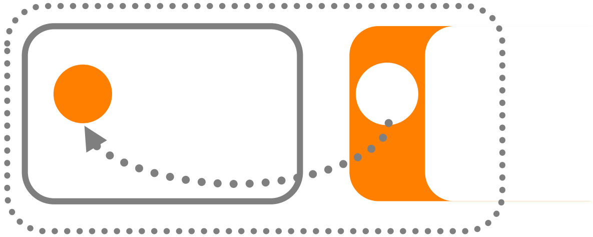
```
a set: #voted to: true for: u
```

(effective behavior of `activityAccessor`)

Activity Contexts: State Scoping

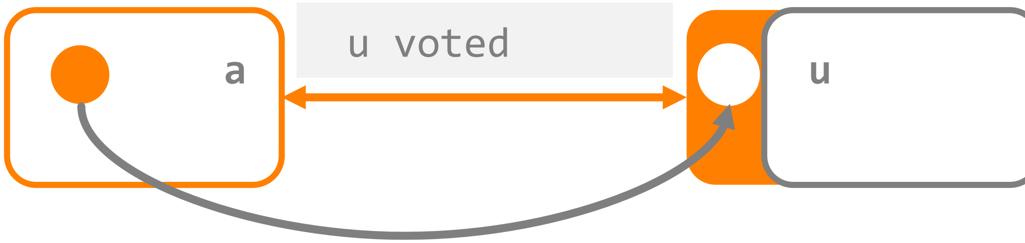
Platform objects may be immutable, where do we store state?

State remains (lexically) scoped to the activity



```
Ballot >> User >> eligible
  <activityAccessor>
  ^false
```

```
Ballot >> User >> voted
  <activityAccessor>
  ^false
```



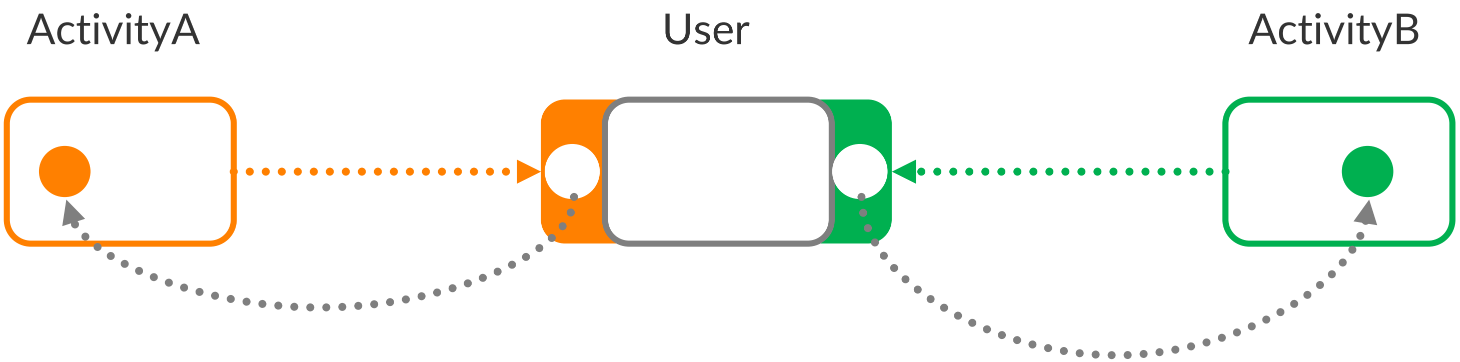
```
a get: #voted for: u
```

(effective behavior of activityAccessor)

Activity Contexts: Names

```
ActivityA >> User >> eligible
  <activityAccessor>
  ^false
```

```
ActivityB >> User >> eligible
  <activityAccessor>
  ^false
```

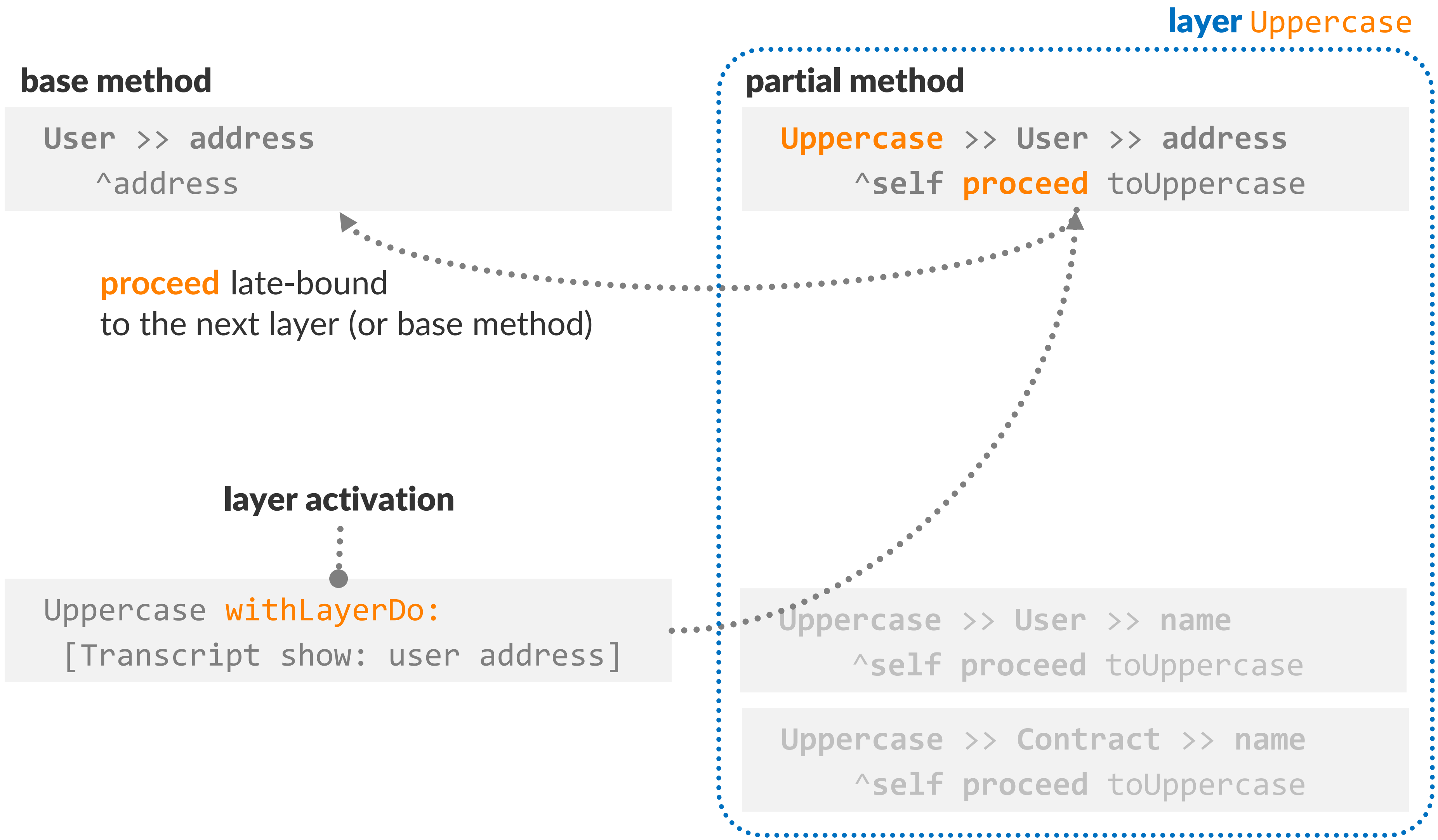


Activities can re-use the same name, but always see their **own** state.

critical, since code is independently developed

`eligible` has no meaning outside an activity.

Recap: Layer-based COP



layer Uppercase

base method

```
User >> address
   ^address
```

partial method

```
Uppercase >> User >> address
     ^self proceed toUppercase
```

proceed late-bound
to the next layer (or base method)

layer activation

```
Uppercase withLayerDo:
 [Transcript show: user address]
```

```
Uppercase >> User >> name
     ^self proceed toUppercase
```

```
Uppercase >> Contract >> name
     ^self proceed toUppercase
```

Activity Contexts vs. Layers

- Activity Contexts are **objects**
  - Identity, state, behavior
  - Communicating via messages

- Activity Contexts are **layers**
  - Partial state/behavior for other objects
  - Cross-cutting (adapts multiple objects/classes at once)
  - Run-time activation and composition

- Subtle differences
  - State per activity (neither layer, nor layered object)
  - Composable with layers, but not other ACs (i.e., no `proceed`/`next` between activities)

Layers within Activities

» Can we exploit composability of layers (and Activity Contexts) to further improve contract code?

Role (of a user)

State (of the activity)

```
Ballot >> initialize
    "contract constructor"
    sender isPollLeader: true.
```

```
Ballot >> startPoll
    self assert: sender isPollLeader.
    self open: true.
```

```
Ballot >> vote: id
    self assert: self open.
    "check if enlisted and not voted"
    self assert: sender canVote.
    "update state"
    (self proposals at: id) addVote.
    sender voted: true.
```

Roles as Layers

- Replace role checks by layer with role-specific behavior

```
Ballot >> initialize
  "contract constructor"
  sender isPollLeader: true.
```

```
Ballot >> startPoll
  self assert: sender isPollLeader.
  self open: true.
```

```
Ballot >> initialize
  "contract constructor"
  sender attach: PollLeader
```

activate Layer at instance

```
PollLeader >> Ballot >> startPoll
  self open: true.
```

Layer definition
(startPoll invisible outside)

State as Layers

- Replace state checks by layer with state-specific behavior

```
Ballot >> startPoll
  self assert: sender isPollLeader.
  self open: true.
```

```
Ballot >> vote: id
  self assert: self open.
  [...] "check if enlisted and not voted"
  [...] "update state"
```

```
Ballot >> startPoll
  self assert: sender isPollLeader.
  self attach: PollOpen.
```

activate Layer at activity

```
PollOpen >> Ballot >> vote: id
  [...] "check if enlisted and not voted"
  [...] "update state"
```

Layer definition
(`vote:` invisible outside)

Layers in Smart Contracts

Traditional contract

```
Ballot >> initialize
   "contract constructor"
   sender isPollLeader: true.
```

```
Ballot >> startPoll
   self assert: sender isPollLeader.
   self open: true.
```

```
Ballot >> vote: id
   self assert: self open.
   "check if enlisted and not voted"
   self assert: sender canVote.
   "update state"
   (self proposals at: id) addVote.
   sender voted: true.
```

Roles and state as layer

```
Ballot >> initialize
   "contract constructor"
   sender attach: PollLeader
```

```
PollLeader >> Ballot >> startPoll

   self attach: PollOpen.
```

```
PollOpen >> Ballot >> vote: id

   "check if enlisted and not voted"
   self assert: sender canVote.
   "update state"
   (self proposals at: id) addVote.
   sender voted: true.
```

Layer Activation Mechanisms in Use

layer activation scoped to **specific instance** (sender "sees" layer whenever control flow enters its scope)

```
Ballot >> initialize
    "contract constructor"
  sender attach: PollLeader
```

```
PollLeader >> Ballot >> startPoll

  self attach: PollOpen.
```

activation during **control flow**

```
SomeLayer withLayerDo: [...]
```

global activation

```
SomeLayer activate.
```

Limitations and Outlook

- **Tooling:** Arrange code in a useful way

- **Use cases:** Explore additional smart contract types
  - (Blind) Double auctions
  - Decentralized Market places
  - Supply chain ledgers
  - …

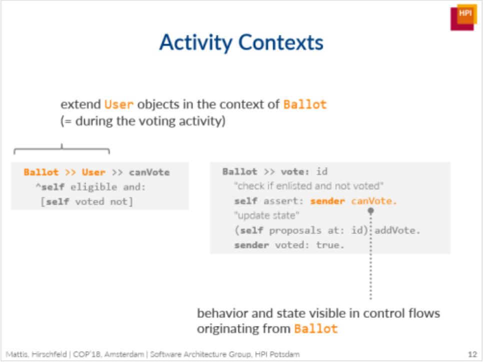
- **Integration:** Explore how to target existing smart contract platforms (e.g. EVM on the Ethereum Blockchain)

Summary

- Activity Contexts have **layer and object** personalities

- ACs are a tool to **decompose large mediators**, such as smart contracts, back into smaller responsibilities
  - Restore encapsulation
  - Scope extensions to activity only

- **Layers** integrate with ACs and can provide further modularity



Activity Contexts

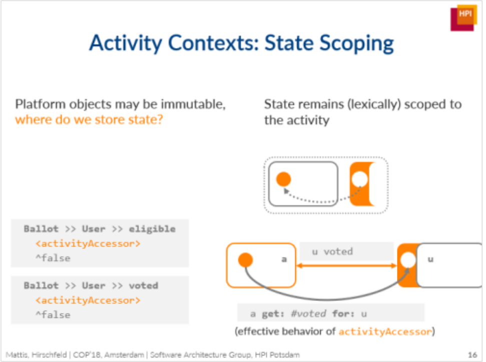
extend User objects in the context of Ballot
(= during the voting activity)

```
Ballot >> User >> canVote
  ^self eligible and:
   [self voted not]
```

```
Ballot >> vote: id
  "check if enlisted and not voted"
  self assert: sender canVote.
  "update state"
  (self proposals at: id) addVote.
  sender voted: true.
```

behavior and state visible in control flows originating from Ballot

Mattis, Hirschfeld | COP'18, Amsterdam | Software Architecture Group, HPI Potsdam 12



Activity Contexts: State Scoping

Platform objects may be immutable, where do we store state?

State remains (lexically) scoped to the activity

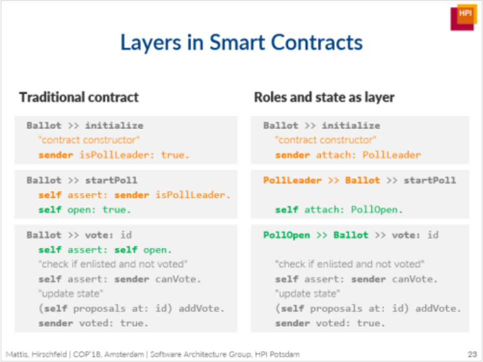
```
Ballot >> User >> eligible
  <activityAccessor>
  ^false
```

```
Ballot >> User >> voted
  <activityAccessor>
  ^false
```

```
a get: #voted for: u
```

(effective behavior of activityAccessor)

Mattis, Hirschfeld | COP'18, Amsterdam | Software Architecture Group, HPI Potsdam 16



Layers in Smart Contracts

Traditional contract

```
Ballot >> initialize
  "contract constructor"
  sender isPollLeader: true.
```

```
Ballot >> startPoll
  self assert: sender isPollLeader.
  self open: true.
```

```
Ballot >> vote: id
  self assert: self open.
  "check if enlisted and not voted"
  self assert: sender canVote.
  "update state"
  (self proposals at: id) addVote.
  sender voted: true.
```

Roles and state as layer

```
Ballot >> initialize
  "contract constructor"
  sender attach: PollLeader
```

```
PollLeader >> Ballot >> startPoll

  self attach: PollOpen.
```

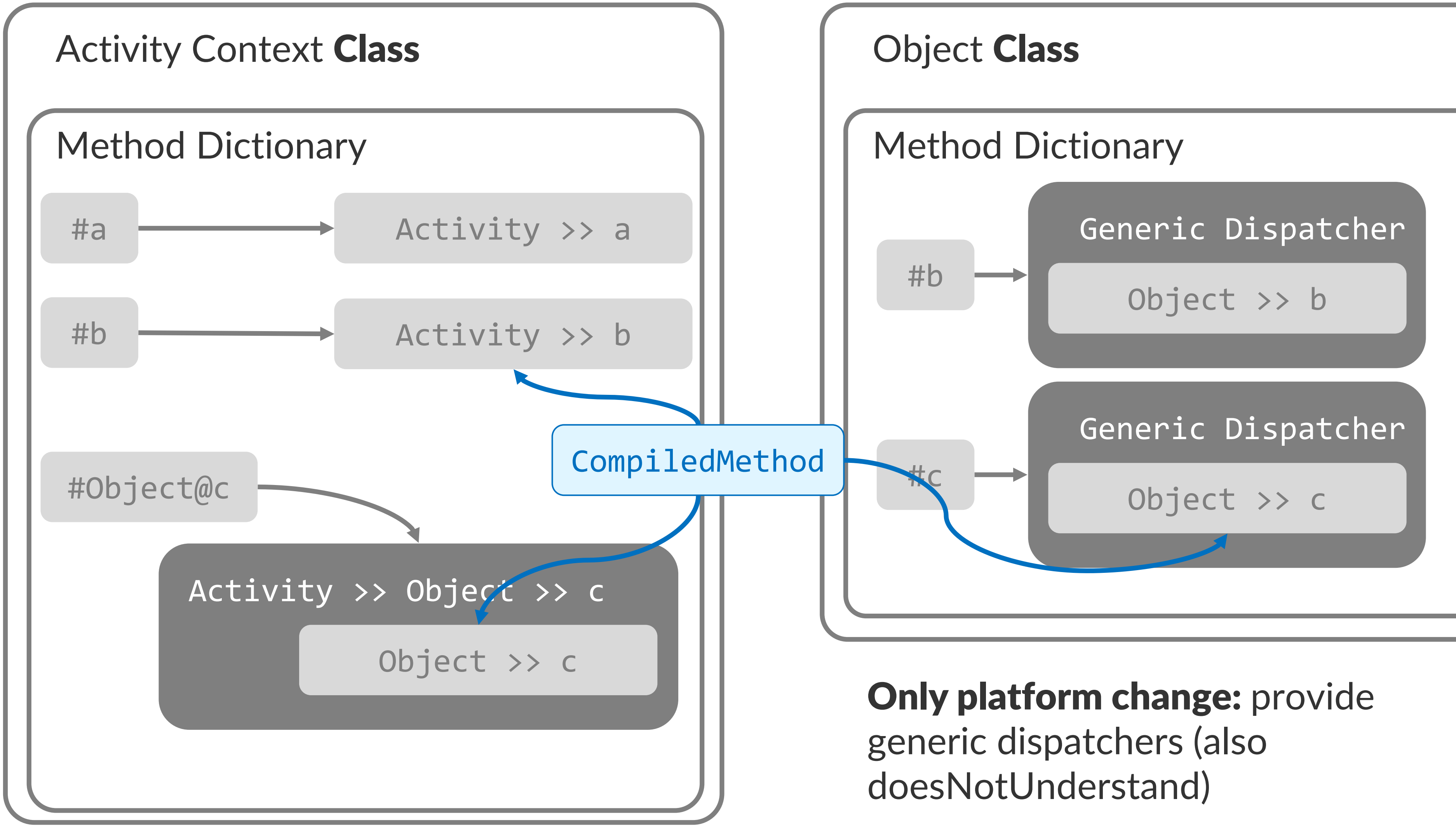
```
PollOpen >> Ballot >> vote: id

  "check if enlisted and not voted"
  self assert: sender canVote.
  "update state"
  (self proposals at: id) addVote.
  sender voted: true.
```

Mattis, Hirschfeld | COP'18, Amsterdam | Software Architecture Group, HPI Potsdam 23

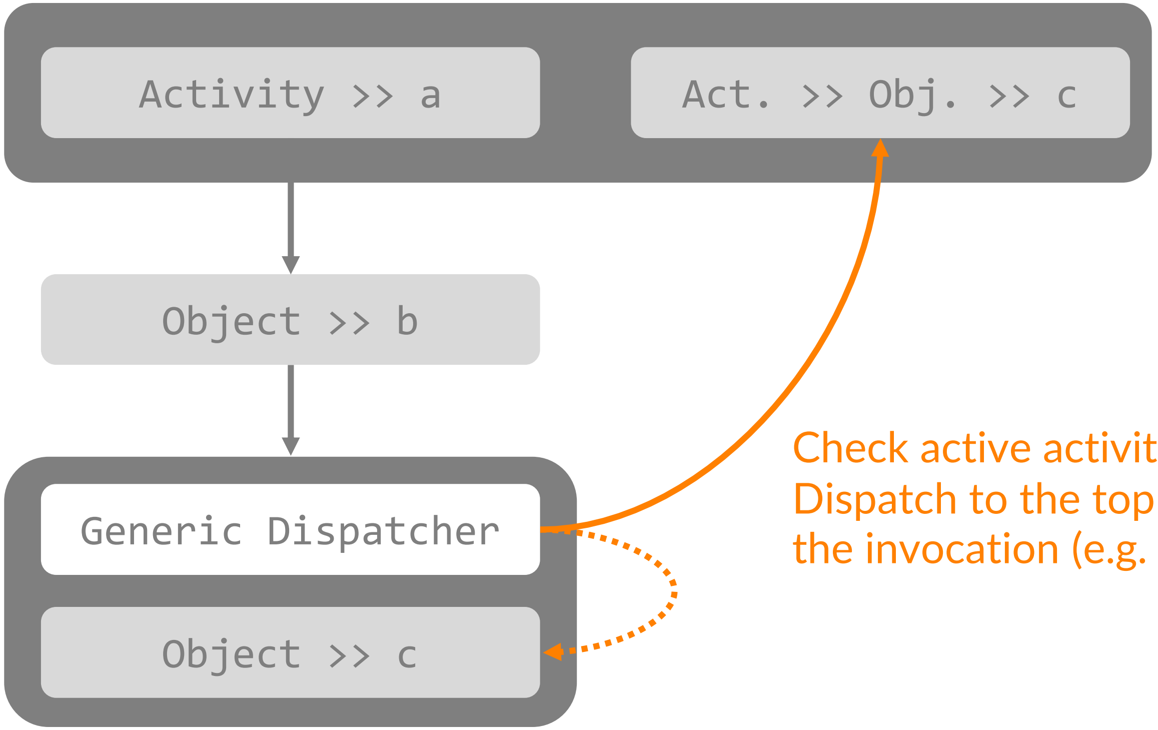
Backup Slides

Implementation



Only platform change: provide generic dispatchers (also doesNotUnderstand)

Implementation



Check active activities on call stack. Dispatch to the top-most that handles the invocation (e.g. `#Object@c`)