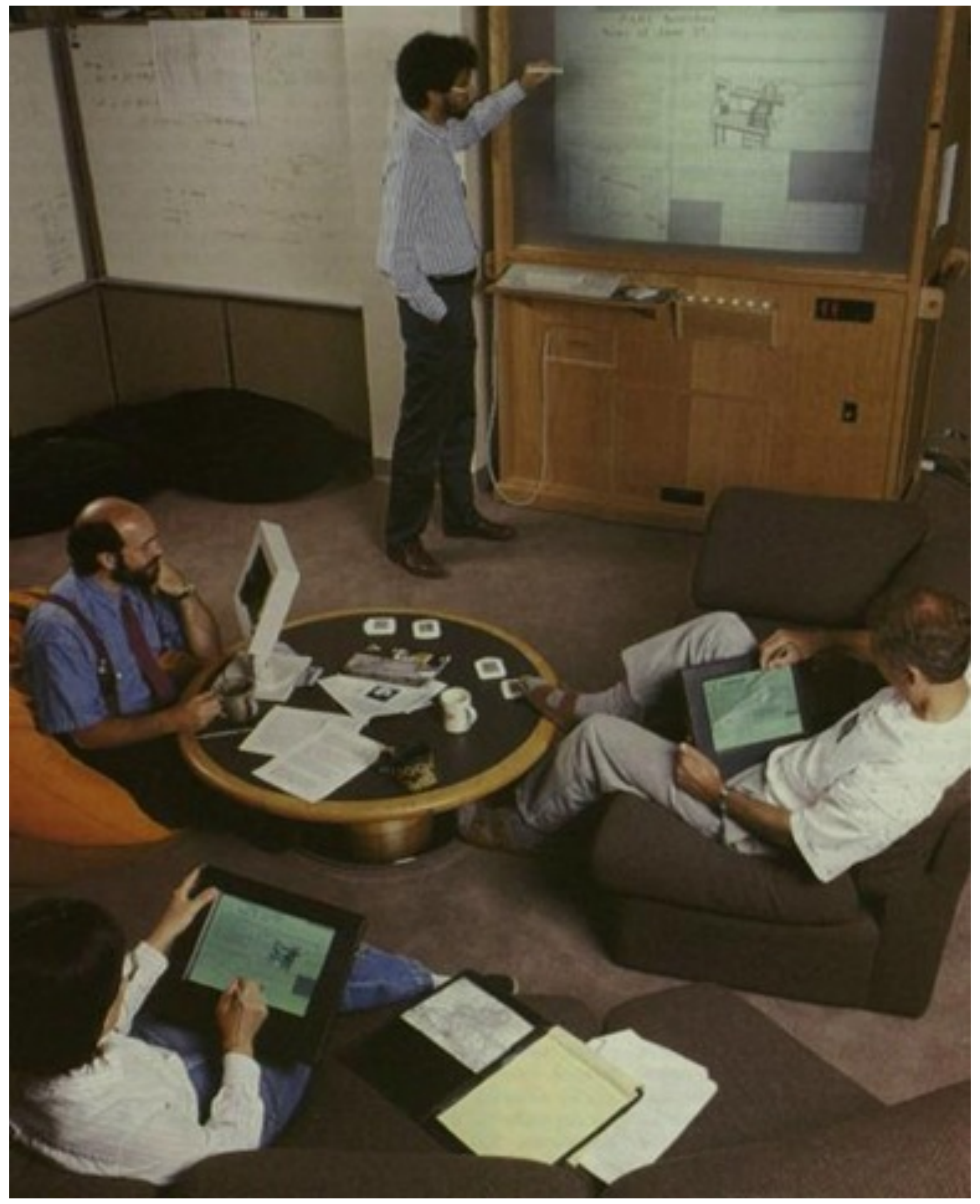
Implementing, Verifying and Debugging Distributed Systems

Elisa Gonzalez Boix

https://soft.vub.ac.be/disco/

How it all started?

- A programming model for ubiquitous computing



Ubiquitous Computing
Vision (1988)

In an ubiquitous context, asynchronous communication suits better, why don’t you implement an actor-based language?

Jessie Dedecker

my own language?!?

Actors in JavaPic%

JavaPic% Evaluator

Elisa Gonzalez Boix, Stijn Mostinckx and Tom Van Cutsem
{elgonzal,smostinc,tvcutsem}@vub.ac.be

Programming Technology Lab
Department of Computer Science
Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussels - Belgium
Fax: (+32 2) 629 35 25

November 2003

Abstract

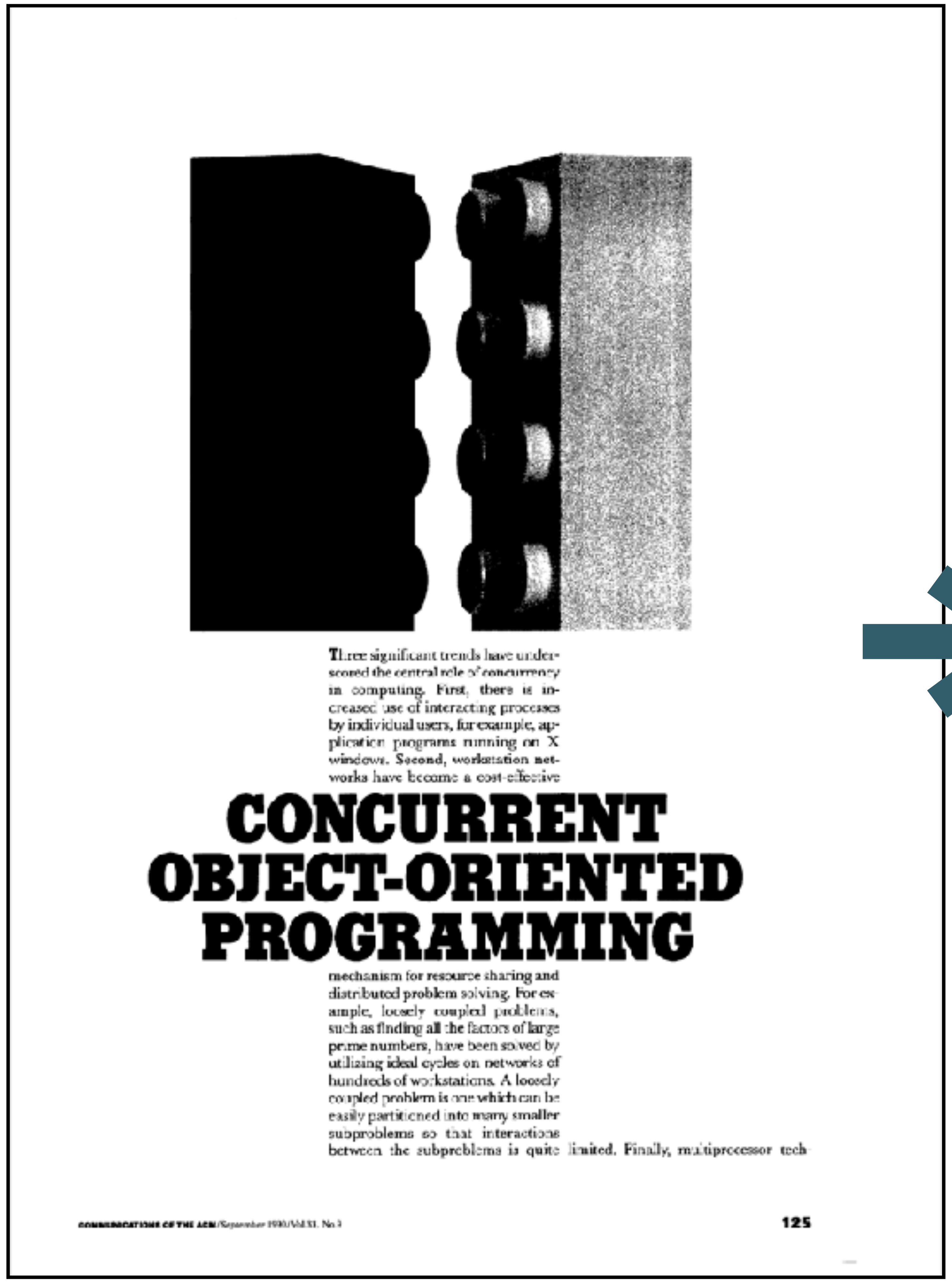
This document accompanies the source code and explains the design and implementation considerations behind the JavaPic% Evaluator.

1 Introduction

The Pico language (D'Hondt, 1996) was originally developed as a teaching environment to provide science students with a programming language easy to read but as powerful and simple as Scheme. However, it also became a research framework for reflective virtual machines and strong mobility (Van Belle and D'Hondt, 2000). Pico was designed as a simple and extensible language based upon quite simple rules. Moreover, all Pico constructs are first class values which implies that basic values, functions, tables, environments and parse trees can be passed as arguments, returned by functions or be bound to a variable.

Pic% was developed as an object oriented extension of the Pico language based on prototype-based language features. This implies that there are no classes, instead it is based on the use of prototypes. New objects are created by cloning existing objects, or by the use of *constructor functions*, which are a transposition of class-based object constructors. Modification and communication is done by message passing. Therefore Pic% supports delegation, cloning of prototypical objects and parent sharing mechanisms. However, Pic% also inherits the simple Pico syntax. Table 1 shows the Pic% syntax. The first three rows are the standard 3X3 syntax rules for Pico (De Meuter et al., 1999) and the last three are specific for Pic%.

1



Three significant trends have underscored the central role of concurrency in computing. First, there is increased use of interacting processes by individual users, for example, application programs running on X windows. Second, workstation networks have become a cost-effective

CONCURRENT OBJECT-ORIENTED PROGRAMMING

mechanism for resource sharing and distributed problem solving. For example, loosely coupled problems, such as finding all the factors of large prime numbers, have been solved by utilizing ideal cycles on networks of hundreds of workstations. A loosely coupled problem is one which can be easily partitioned into many smaller subproblems so that interactions between the subproblems is quite limited. Finally, multiprocessor tech-

COMMUNICATIONS OF THE ACM/September 1990/Vol.33, No.9 125

```
factor(n): createActor ({
  fac (c, client)::{
     if ( n> 1, {
   next: factor(n-1);
        if( n = c, next.fac(c, client),
           next.fac( c*n, client))},
         client.result(c) )
  }
});

factorial : createActor({
  result(res):: { display(res)};
  compute(num):: {
  a : factor(num);
  a.fac (num, thisActor())}
});

factorial.compute(3);
```

How it all started again?

- a programming model for ubiquitous computing

Theo
D’hondt

Wolfgang
De Meuter

what about exploring a failure handling model for AmbientTalk?

AmbientTalk?

Jessie
Dedecker

Tom
Van Cutsem

Stijn
Mostinckx

J. Dedecker, T. Van Cutsem, S. Mostinckx, T. D'Hondt, W. De Meuter. 2006. Ambient-Oriented Programming in AmbientTalk. In Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP), LNCS Vol. 4067, pp. 230-254, Springer-Verlag.”, 2006.

AmbientTalk’s Distributed Model = OO + Events

A superset of object-oriented programming that is explicitly geared towards programming distributed applications that run on mobile ad hoc networks

Generate and receive application requests

```
obj<-msg(arg)
def msg(param) { ... }
```

Follow-up on outstanding requests

```
when: future becomes: { |result| ... }
```

React to services appearing and disappearing

```
when: type discovered: { |ref| ... }
whenever: type discovered: { |ref| ... }
```

React to references disconnecting, reconnecting

```
when: ref disconnected: { ... }
when: ref reconnected: { ... }

whenever: ref disconnected: { ... }
whenever: ref reconnected: { ... }
```

Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, and Wolfgang De Meuter. Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks. In Inter. Conf. of the Chilean Computer Science Society (SCCC), pages 3–12. IEEE Computer Society, 2007.

Distributed Applications in AmbientTalk

https://soft.vub.ac.be/amop/

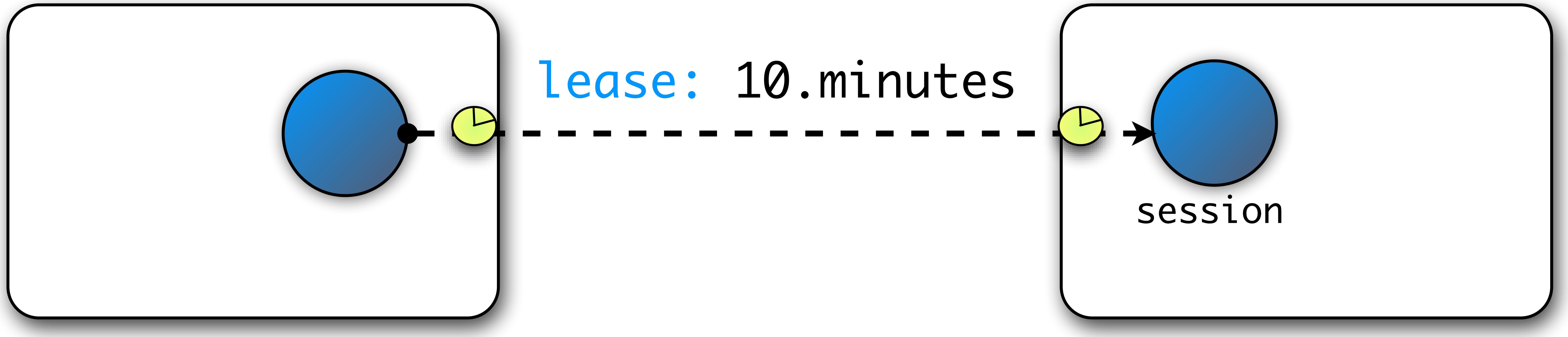
```
deftype PingPong;

def pingPong := object: {
  def ping(){
     system.println(“received ping");
    `pong;
  }
};

export: pingPong as: PingPong;
```

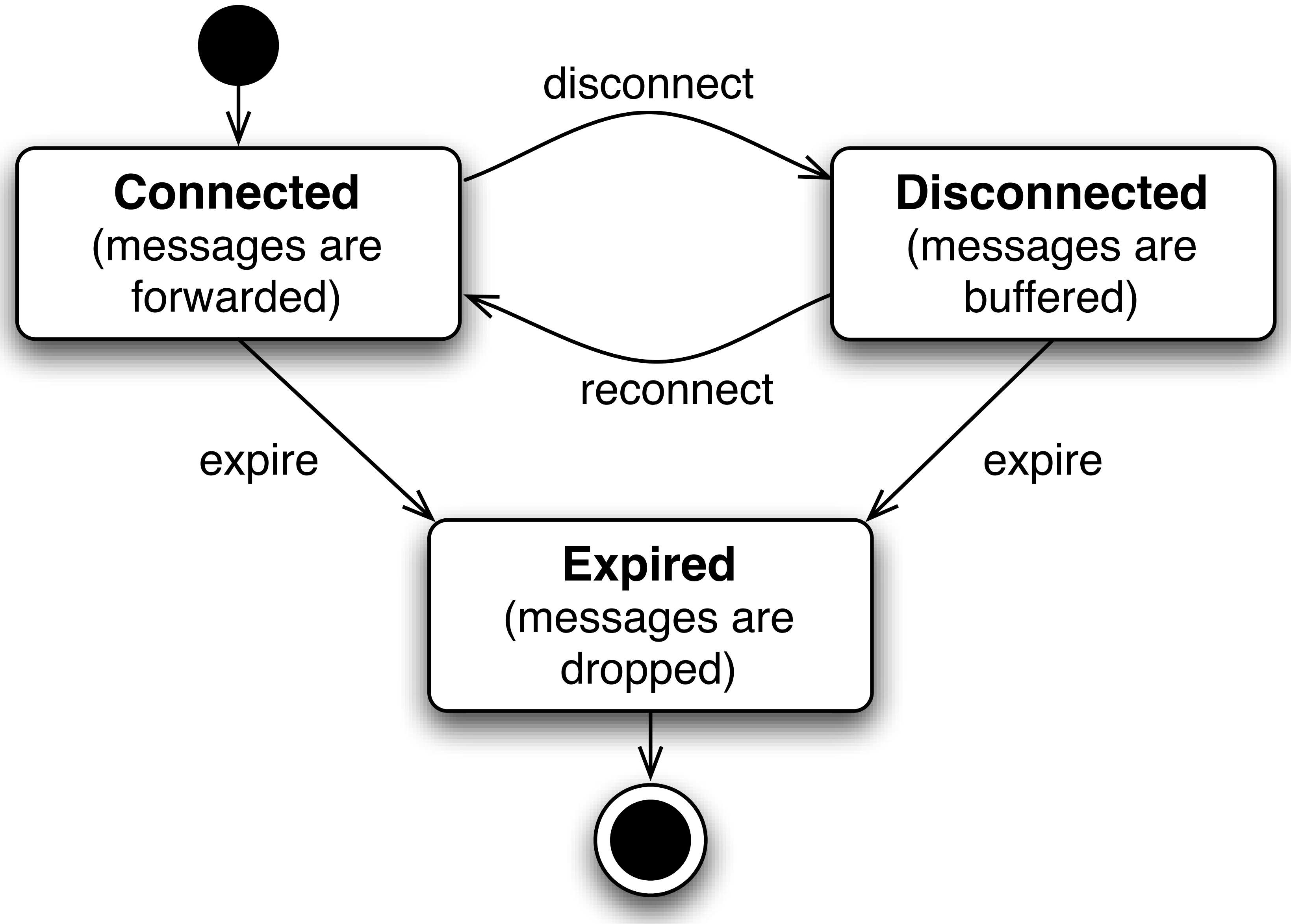
```
deftype PingPong;

whenever: PingPong
discovered: {
    |farRef|
     when: farRef<-ping()@FutureMessage()
     becomes: {
       |val|
       system.println("received " + val);
     }
}
```

- Data stored at the owner, and all operations go via asynchronous message passing

How to reconcile failures with distributed event-based model?

"A lease denotes the right to access a resource for a specific duration negotiated when the access is first requested."



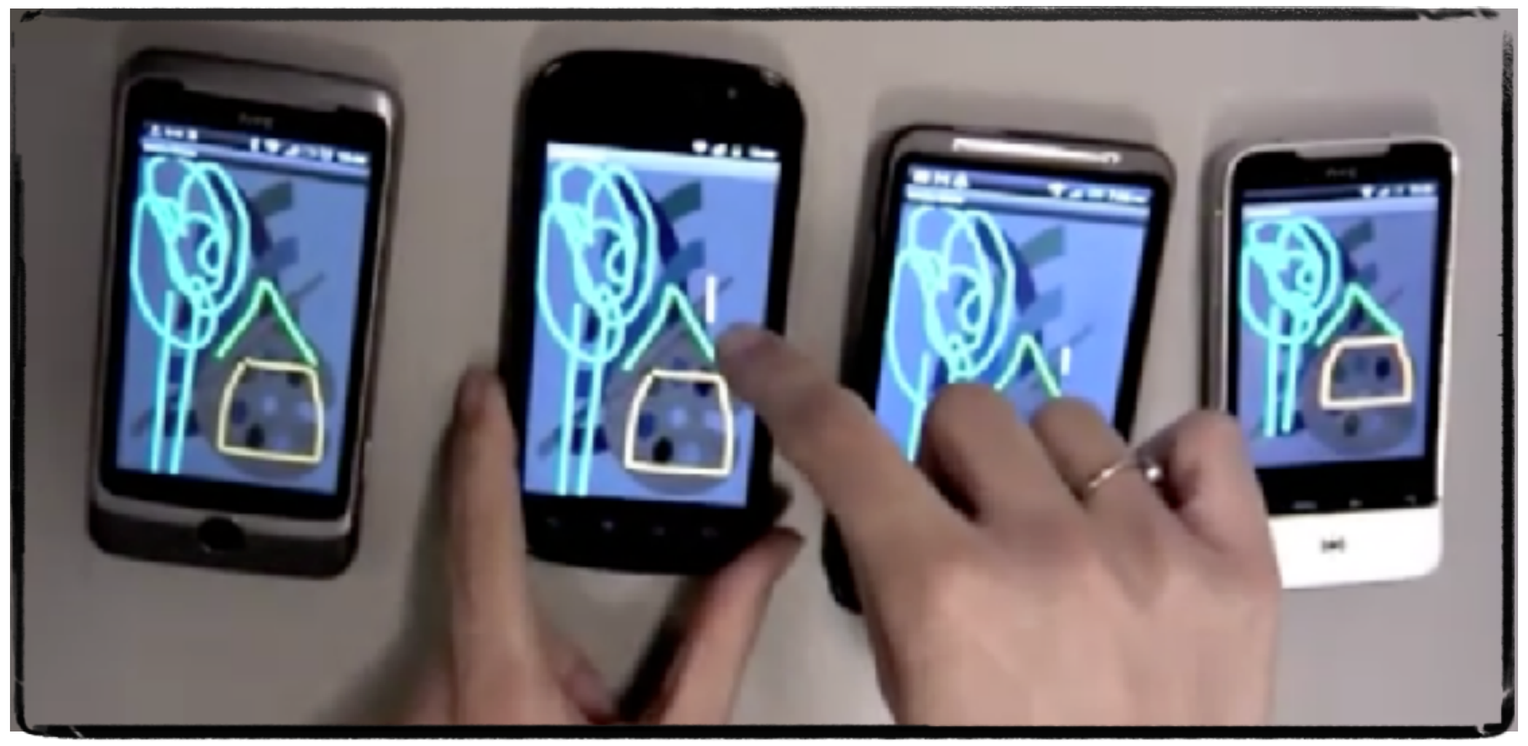



```
when: ref disconnected: { ... }
when: ref reconnected: { ... }
when: ref expired: { ... }

whenever: ref disconnected: { ... }
whenever: ref reconnected: { ... }
```

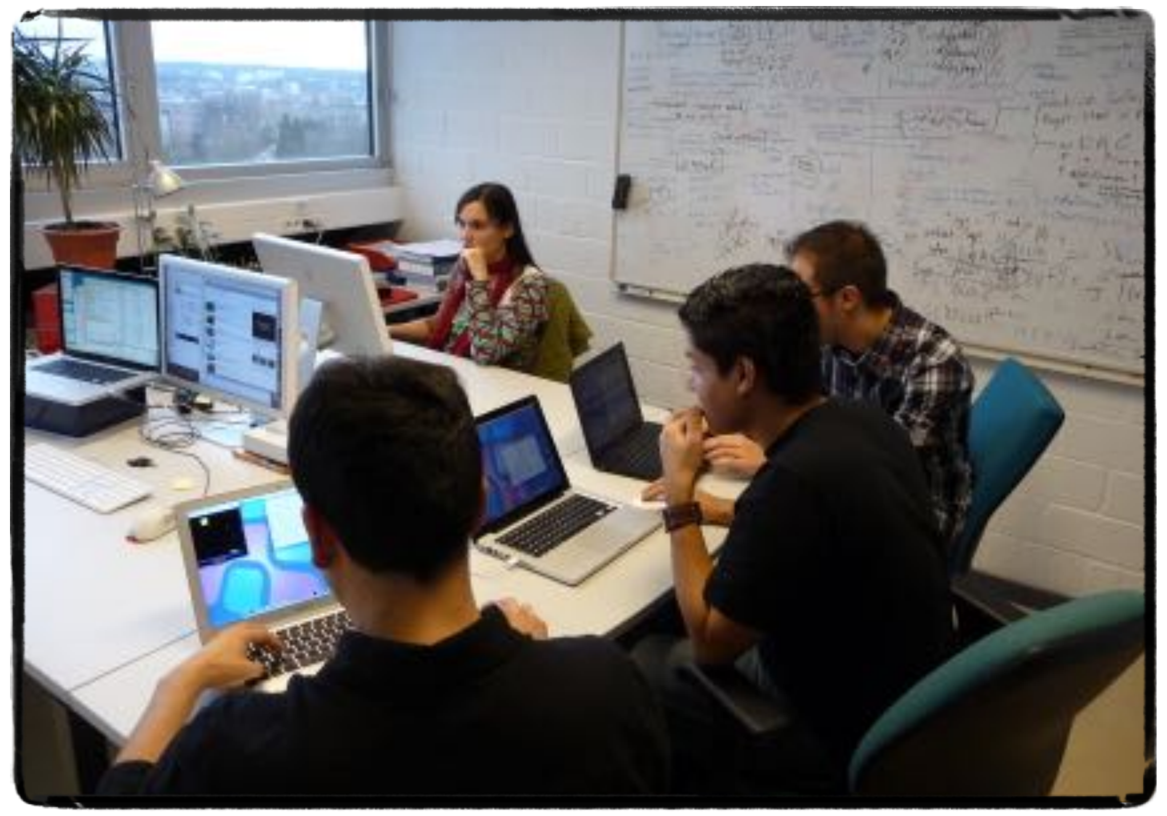
Distributed P2P applications

Urbiflock

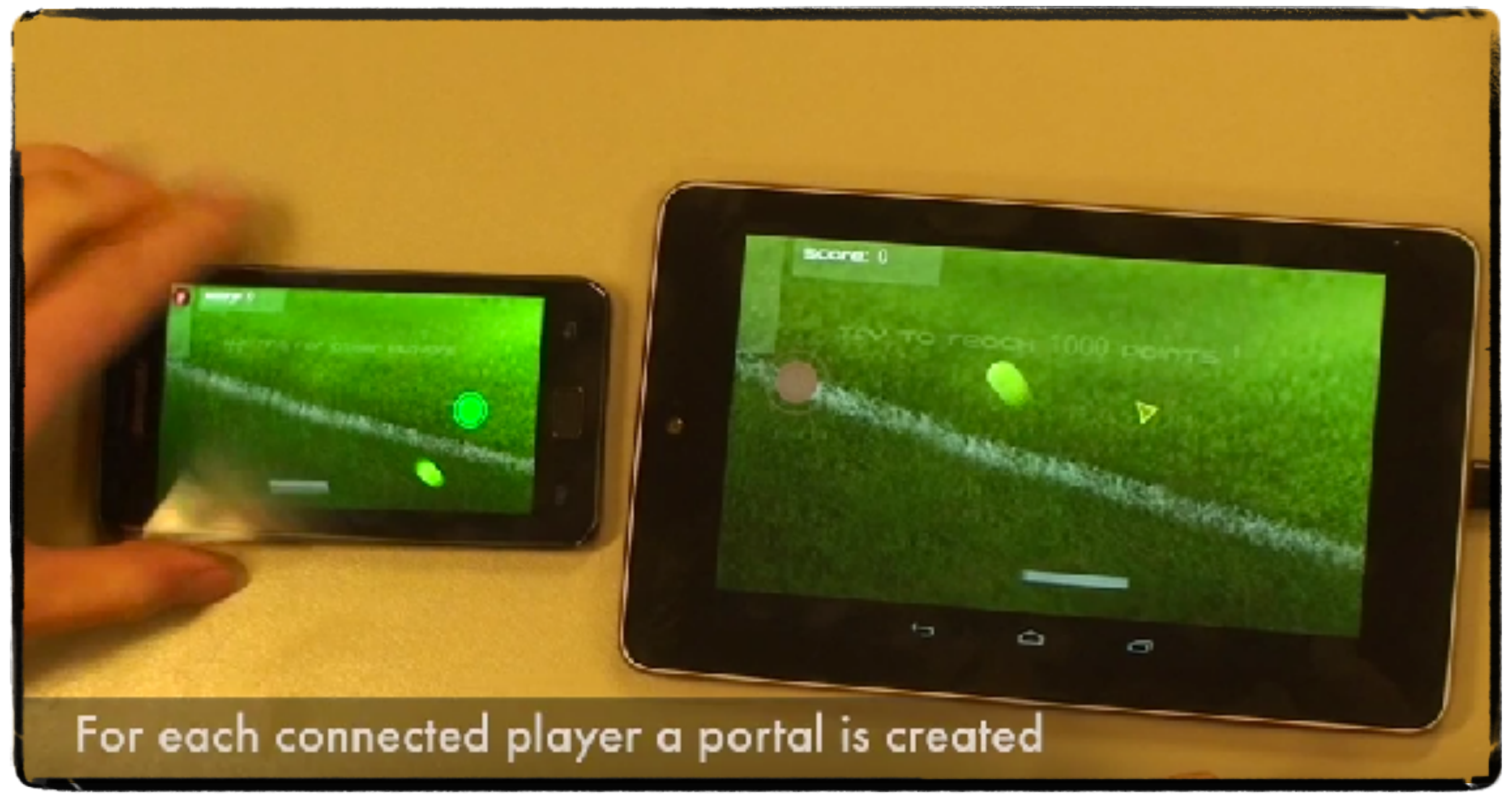
WeScribble



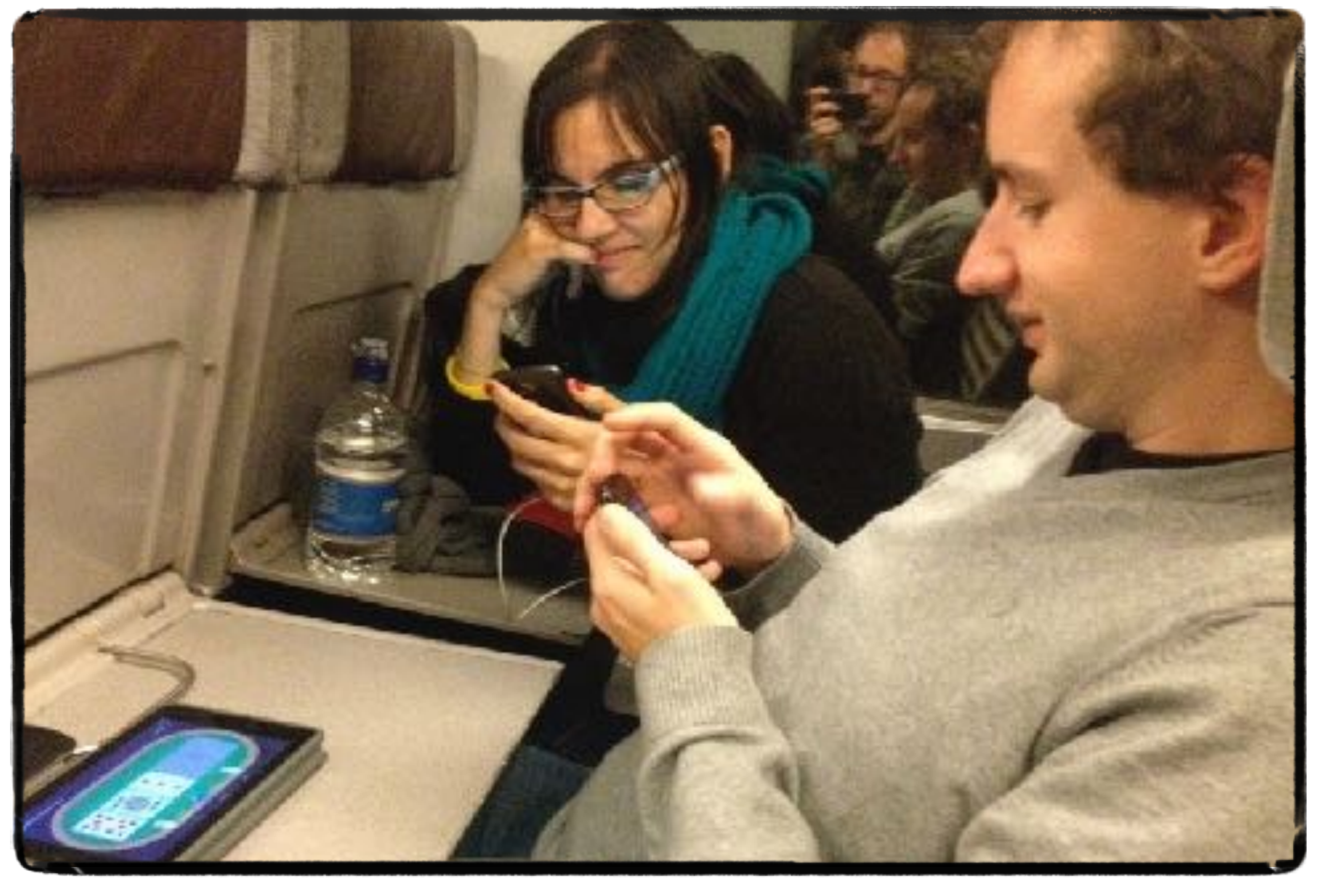
PortalPong



WePong



WePoker



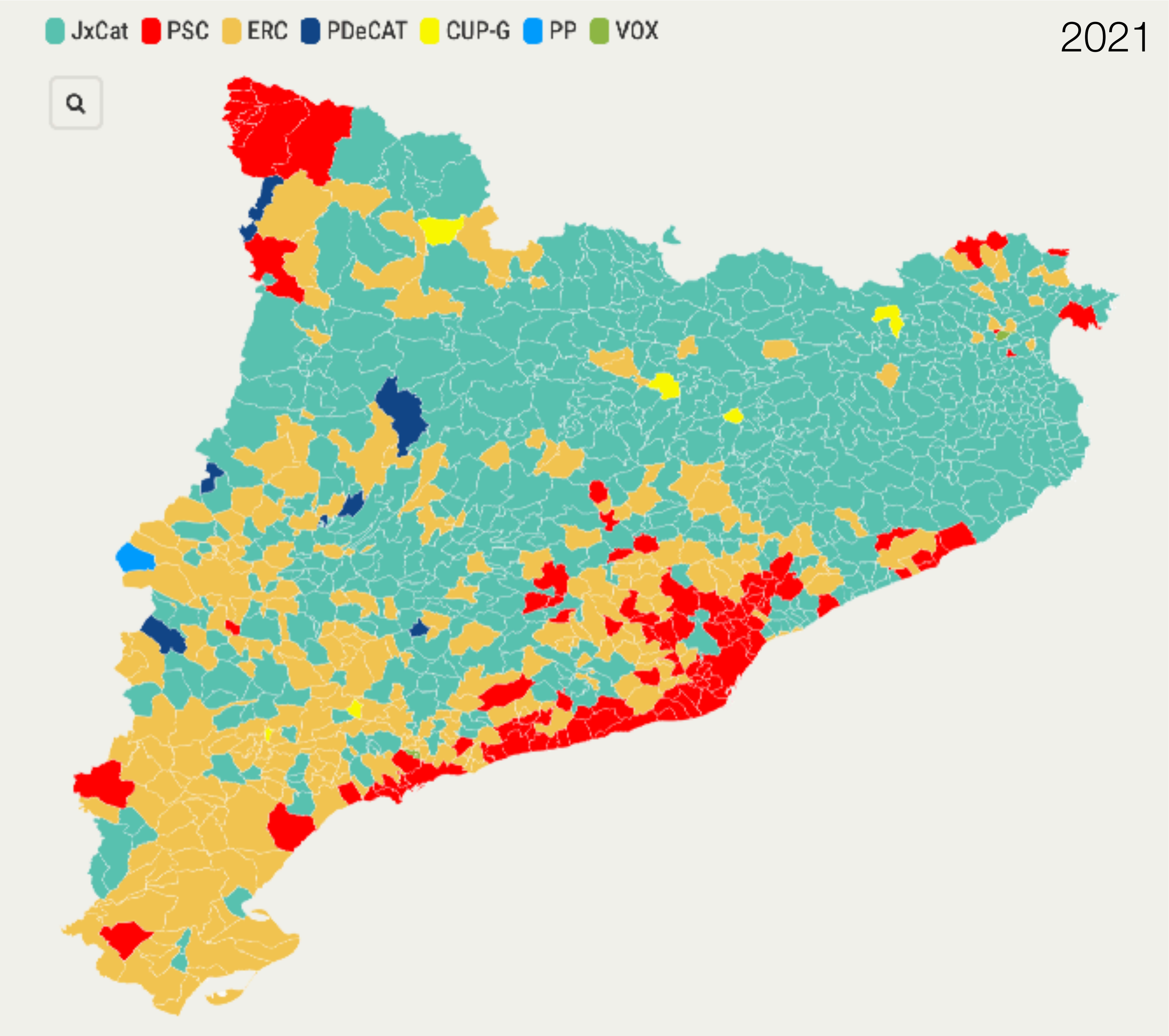
http://tinyurl.com/ambienttalkyoutube

Many times data needed to be shared…


```
def counter := isolate: {

    def val := 0;

    def incr(){ val := val + 1 };
    def decr(){ val := val - 1 };
    def value(){ val };
};
```

- ✔ increasing availability: operations can execute locally
- ✔ improving reliability: avoid single points of failure.

- ✖ providing some form of consistency is left to the application programmer



Could we build replicated objects so that ..

- developers can customize conflict resolution according to the application's needs

```
Bayou_Write(
  update = {insert, Meetings, 12/18/95, 10:00am, 60min, Project Meeting: Kevin},
  dependency_check = {
    query = SELECT key FROM Meetings WHERE day = 12/18/95
             AND start < 11:00am AND end > 10:00am,
    expected_result = EMPTY },
  mergeproc = {
    alternates = {12/18/95, 12:00pm};
    newupdate = {};
    FOREACH a IN alternates {
      # check if there would be a conflict
      IF (NOT EMPTY (
        SELECT key FROM Meetings WHERE day = a.date
          AND start < a.time + 60min AND end > a.time))
        CONTINUE;
      # no conflict, can schedule meeting at that time
      newupdate = {insert, Meetings, a.date, a.time, 60min, Project Meeting: Kevin};
      BREAK;
    }
    IF (newupdate = {}) # no alternate is acceptable
       newupdate = {insert, ErrorLog, 12/18/95, 10:00am, 60min, Project Meeting: Kevin};
    RETURN newupdate;
  }
)
```

The Bayou Architecture: Support for Data Sharing among Mobile Users

Alan Demers, Karin Petersen, Mike Spreitzer, Douglas Terry, Marvin Theimer, Brent Welch

Computer Science Laboratory
Xerox Palo Alto Research Center
Palo Alto, California 94304 U.S.A.
contact: terry@parc.xerox.com

Abstract

The Bayou System is a platform of replicated, highly-available, variable-consistency, mobile databases on which to build collaborative applications. This paper presents the preliminary system architecture along with the design goals that influenced it. We take a fresh, bottom-up and critical look at the requirements of mobile computing applications and carefully pull together both new and existing techniques into an overall architecture that meets these requirements. Our emphasis is on supporting application-specific conflict detection and resolution and on providing application-controlled inconsistency.

1. Introduction

The Bayou project at Xerox PARC has been designing a system to support data sharing among mobile users. The system is intended to run in a mobile computing environment that includes portable machines with less than ideal network connectivity. In particular, a user's computer may have a wireless communication device, such as a cell modem or packet radio transceiver relying on a network infrastructure that is not universally available and perhaps unreasonably expensive. It may use short-range line-of-sight communication, such as the infrared "beaming" ports available on some commercial personal digital assistants (PDAs). Alternatively, the computer may have a conventional modem requiring it to be physically connected to a phone line when sending and receiving data or may only be able to communicate with the rest of the system when inserted in a docking station. Finally, its only communication device may be a diskette that is transported between machines by humans. The main characteristic of these communication capabilities is that a mobile computer may experience extended and sometimes involuntary disconnection from many or all of the other devices with which it wants to share data.

We believe that mobile users want to share their appointment calendars, bibliographic databases, meeting notes, evolving design documents, news bulletin boards, and other types of data in spite of their intermittent network connectivity. The focus of the Bayou project has been on exploring mechanisms that let mobile clients actively read and write shared data. Even though the system must cope with both voluntary and involuntary communication outages, it should look to users, to the extent possible, like a centralized, highly-available database service. This paper presents detailed goals for the overall system architecture and discusses the design decisions that we made to meet these goals.

2. Architectural design decisions

Goal: Support for portable computers with limited resources.

Design: A flexible client-server architecture.

Many of the devices that we envision being commonly used, such as PDAs and the ParcTab developed within our lab [24], have insufficient storage for holding copies of all, or perhaps any, of the data that their users want to access. For this reason, our architecture is based on a division of functionality between *servers*, which store data, and *clients*, which read and write data managed by servers. A server is any machine that holds a complete copy of one or more *databases*. We use the term "database" loosely to denote a collection of data items; whether such data is managed as a relational database or simply stored in a conventional file system is left unspecified in the architecture. Clients are able to access data residing on any server to which they can communicate, and conversely, any machine holding a copy of a database, including personal laptops, should be willing to service read and write requests from other nearby machines.

Could we build replicated objects so that ..

- developers can customize conflict resolution according to the application's needs

- without exposing them to merge procedures

```
Bayou_Write(
  update = {insert, Meetings, 12/18/95, 10:00am, 60min, Project Meeting: Kevin},
  dependency_check = {
    query = SELECT key FROM Meetings WHERE day = 12/18/95
             AND start < 11:00am AND end > 10:00am,
    expected_result = EMPTY },
  mergeproc = {
    alternates = {12/18/95, 12:00pm};
    newupdate = {};
    FOREACH a IN alternates {
      # check if there would be a conflict
      IF (NOT EMPTY (
        SELECT key FROM Meetings WHERE day = a.date
          AND start < a.time + 60min AND end > a.time))
        CONTINUE;
      # no conflict, can schedule meeting at that time
      newupdate = {insert, Meetings, a.date, a.time, 60min, Project Meeting: Kevin};
      BREAK;
    }
    IF (newupdate = {}) # no alternate is acceptable
       newupdate = {insert, ErrorLog, 12/18/95, 10:00am, 60min, Project Meeting: Kevin};
    RETURN newupdate;
  }
)
```

The Bayou Architecture: Support for Data Sharing among Mobile Users

Alan Demers, Karin Petersen, Mike Spreitzer, Douglas Terry, Marvin Theimer, Brent Welch

Computer Science Laboratory
Xerox Palo Alto Research Center
Palo Alto, California 94304 U.S.A.
contact: terry@parc.xerox.com

Abstract

The Bayou System is a platform of replicated, highly-available, variable-consistency, mobile databases on which to build collaborative applications. This paper presents the preliminary system architecture along with the design goals that influenced it. We take a fresh, bottom-up and critical look at the requirements of mobile computing applications and carefully pull together both new and existing techniques into an overall architecture that meets these requirements. Our emphasis is on supporting application-specific conflict detection and resolution and on providing application-controlled inconsistency.

1. Introduction

The Bayou project at Xerox PARC has been designing a system to support data sharing among mobile users. The system is intended to run in a mobile computing environment that includes portable machines with less than ideal network connectivity. In particular, a user's computer may have a wireless communication device, such as a cell modem or packet radio transceiver relying on a network infrastructure that is not universally available and perhaps unreasonably expensive. It may use short-range line-of-sight communication, such as the infrared "beaming" ports available on some commercial personal digital assistants (PDAs). Alternatively, the computer may have a conventional modem requiring it to be physically connected to a phone line when sending and receiving data or may only be able to communicate with the rest of the system when inserted in a docking station. Finally, its only communication device may be a diskette that is transported between machines by humans. The main characteristic of these communication capabilities is that a mobile computer may experience extended and sometimes involuntary disconnection from many or all of the other devices with which it wants to share data.

We believe that mobile users want to share their appointment calendars, bibliographic databases, meeting notes, evolving design documents, news bulletin boards, and other types of data in spite of their intermittent network connectivity. The focus of the Bayou project has been on exploring mechanisms that let mobile clients actively read and write shared data. Even though the system must cope with both voluntary and involuntary communication outages, it should look to users, to the extent possible, like a centralized, highly-available database service. This paper presents detailed goals for the overall system architecture and discusses the design decisions that we made to meet these goals.

2. Architectural design decisions

Goal: Support for portable computers with limited resources.

Design: A flexible client-server architecture.

Many of the devices that we envision being commonly used, such as PDAs and the ParcTab developed within our lab [24], have insufficient storage for holding copies of all, or perhaps any, of the data that their users want to access. For this reason, our architecture is based on a division of functionality between *servers*, which store data, and *clients*, which read and write data managed by servers. A server is any machine that holds a complete copy of one or more *databases*. We use the term "database" loosely to denote a collection of data items; whether such data is managed as a relational database or simply stored in a conventional file system is left unspecified in the architecture. Clients are able to access data residing on any server to which they can communicate, and conversely, any machine holding a copy of a database, including personal laptops, should be willing to service read and write requests from other nearby machines.

Replicated Data Types (RDTs)

Specification 6 State-based increment-only counter (vector version)

```
payload integer[n] P
    initial [0, 0, . . . , 0]
update increment ()
    let g = myID()
    P[g] := P[g] + 1
query value () : integer v
    let v = ∑_i P[i]
compare (X, Y) : boolean b
    let b = (∀i ∈ [0, n − 1] : X.P[i] ≤ Y.P[i])
merge (X, Y) : payload Z
    let ∀i ∈ [0, n − 1] : Z.P[i] = max(X.P[i], Y.P[i])
```

Shapiro et al. 2011

Replicated Data Types (RDTs)

Specification 6 State-based increment-only counter (vector version)

```
payload integer[n] P
    initial [0,0,...,0]
update increment ()
    let g = myID()
    P[g] := P[g] + 1
query value () : integer v
    let v = ∑_i P[i]
compare (X, Y) : boolean b
    let b = (∀i ∈ [0,n−1] : X.P[i] ≤ Y.P[i])
merge (X, Y) : payload Z
    let ∀i ∈ [0,n−1] : Z.P[i] = max(X.P[i],
```

Shapiro et al. 2011

RDT

```
def statedBasedCounter := object: {

  def vInc;
  def myId;

  def init(typeName,id,n) { .. };
  def increment() {
    def val := vInc.at(myId);
    vInc.atPut(myId, val +1);
  };
  def value() {
    def res := 0;
    vInc.each: { |val|
      res := res + val };
    res
  };
  def merge(senderVector) {
    def i := 0;
    vInc.each: { |a|
      def b := senderVector.get(i);
      vInc.atPut(i, Math.max(a,b));
      i := i + 1}};
… };
```

Replicated Data Types (RDTs)

Specification 6 State-based increment-only counter (vector version)

payload integer$[n]$ P
    initial $[0, 0, \ldots, 0]$
update *increment* ()
    let $g = myID()$
    $P[g] := P[g] + 1$
query *value* () : integer v
    let $v = \sum_i P[i]$
compare (X, Y) : boolean b
    let $b = (\forall i \in [0, n-1] : X.P[i] \leq Y.P[i])$
merge (X, Y) : payload Z
    let $\forall i \in [0, n-1] : Z.P[i] = \max(X.P[i],$

Shapiro et al. 2011

RDT Distribution aspects

```
def statedBasedCounter := object: {
  import CRDTModule.CRDTTrait;
  def vInc;
  def myId;

  def init(typeName,id,n) { .. };
  def increment() {
    def val := vInc.at(myId);
    vInc.atPut(myId, val +1);
  };
  def value() {
    def res := 0;
    vInc.each: { |val|
      res := res + val };
    res
  };
  def merge(senderVector) {
    def i := 0;
    vInc.each: { |a|
      def b := senderVector.get(i);
      vInc.atPut(i, Math.max(a,b));
      i := i + 1}};
… };
```

```
def CRDTTrait := object: {

    def typeName := defaultCRDT;
    def replicas := [];

   def sync(){
      self.broadcast(<-merge(self.serialize()));
    };

  def broadcast(msg) {
    self.replicas.each: { |farRef|
      farRef <+ msg
    }
  };

  def goOnline(){
     export: self as: (self.typeName);
     whenever: (self.typeName) discovered: {
       | farRef |
        self.replicas := self.replicas + [farRef];
     }
   }
};
```

Could we build replicated data types that..

- are application-specific ?
    - customize concurrency semantics to the application needs
- support application invariants?
- are correct out-of-the box?
- can be arbitrarily composed?
- can be applied to dynamic environments with memory and network constraints?

ECROs

Simplifying the development of application-specific RDTs



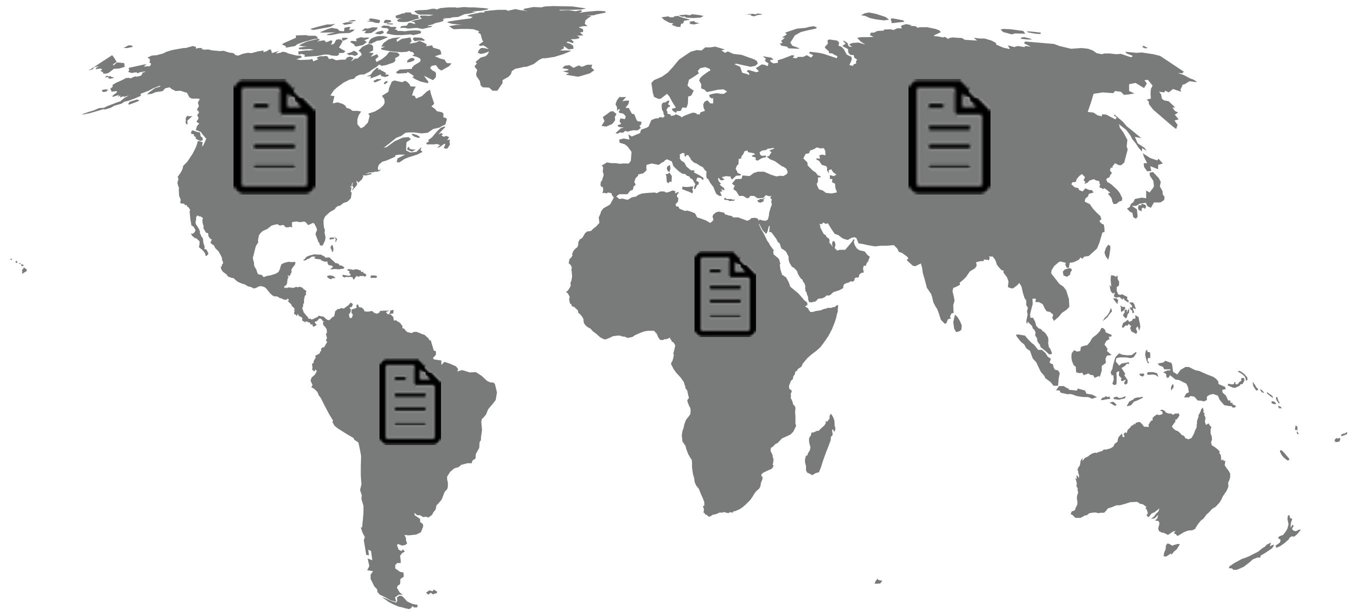
Kevin De Porre, Carla Ferreira, Nuno Preguiça, and Elisa Gonzalez Boix. 2021. ECROs: building global scale systems from sequential code. Proc. ACM Program. Lang. 5, OOPSLA, Article 107 (October 2021), 30 pages. https://doi.org/10.1145/3485484

Explicitly Consistent Replicated Object (ECRO)

- General approach for building hybrid RDTs

sequential
data type

Distributed
Specification



Replicated data type

- Avoids unnecessary coordination

Fast when possible (EC)
consistent when needed (SC)

Explicitly Consistent Replicated Object (ECRO)

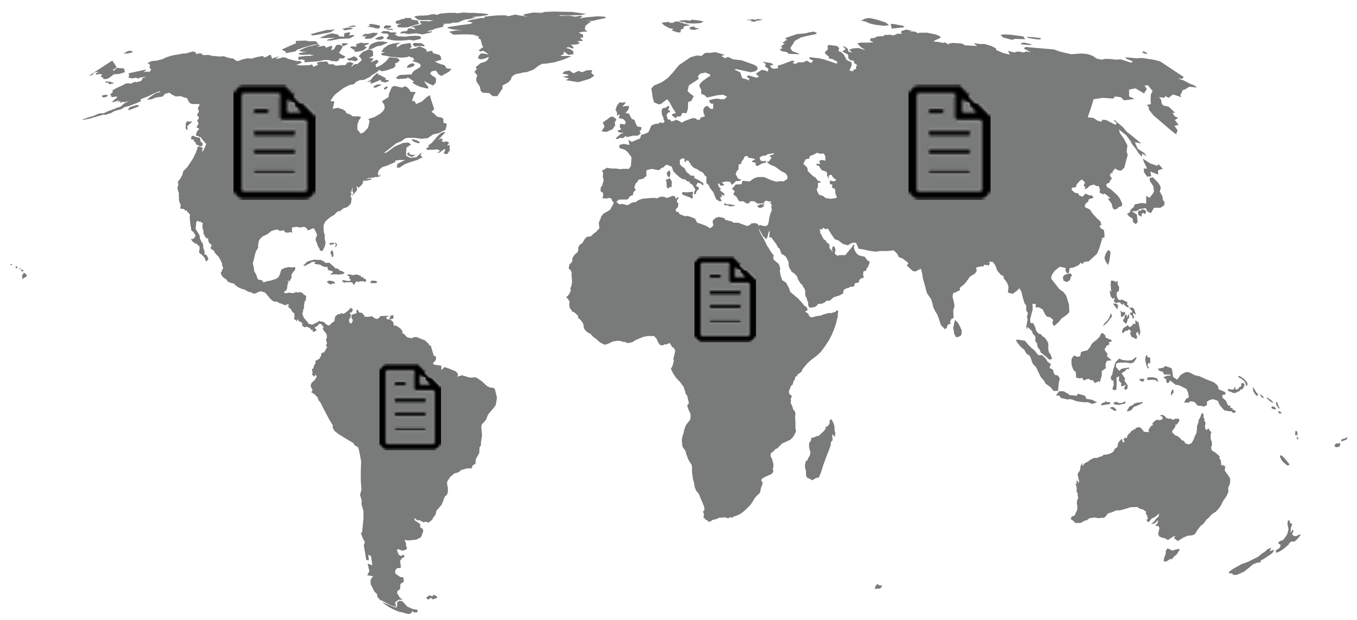
- General approach for building hybrid RDTs

sequential data type

```
case class AWSet[V](set: Set[V]) {
  def add(x: V) =
    new AWSet(set + x)

  def remove(x: V) =
    new AWSet(set - x)

  def contains(x: V) =
    set.contains(x)
}
```

Distributed Specification

```
object AWSet {
  // contains: V × State × Bool
  val contains: Relation = ...

  postcondition of add {
    (old: OldState, res: NewState) =>
      contains(x, res) /\
      contains.copyExcept(old -> res, elem === x)
  }

  postcondition of remove {
    (old: OldState, res: NewState) =>
      not (contains(x, res)) /\
      contains.copyExcept(old -> res, elem === x)
  }

  invariant on add {
    (_: OldState, res: NewState) =>
      contains(x, res)
  }
}
```



Replicated data type

Fast when possible (EC)
consistent when needed (SC)

Building Geo-Distributed Apps, the ECRO Way

Implementing an Add-Wins Set

Sequential implementation in Scala

```scala
case class AWSet[V](set: Set[V]) {
  def add(x: V) =
    new AWSet(set + x)

  def remove(x: V) =
    new AWSet(set - x)

  def contains(x: V) =
    set.contains(x)
}
```

Building Geo-Distributed Apps, the ECRO Way

Implementing an Add-Wins Set

Sequential implementation in Scala

```scala
case class AWSet[V](set: Set[V]) {
  def add(x: V) =
    new AWSet(set + x)

  def remove(x: V) =
    new AWSet(set - x)

  def contains(x: V) =
    set.contains(x)
}
```

DSL for distributed specification

```
object AWSet {
  // contains: V × State × Bool
  val contains: Relation = ...

  postcondition of add {
    (old: OldState, res: NewState) =>
      contains(x, res) /\
      contains.copyExcept(old -> res, elem === x)
  }

  postcondition of remove {
    (old: OldState, res: NewState) =>
      not (contains(x, res)) /\
      contains.copyExcept(old -> res, elem === x)
  }

  invariant on add {
    (_: OldState, res: NewState) =>
      contains(x, res)
  }
}
```

Building Geo-Distributed Apps, the ECRO Way

Implementing an ~~Add-Wins~~ Remove-Wins Set

Sequential implementation in Scala

```scala
case class AWSet[V](set: Set[V]) {
  def add(x: V) =
    new AWSet(set + x)

  def remove(x: V) =
    new AWSet(set - x)

  def contains(x: V) =
    set.contains(x)
}
```

DSL for distributed specification

```
object AWSet {
  // contains: V × State × Bool
  val contains: Relation = ...

  postcondition of add {
    (old: OldState, res: NewState) =>
      contains(x, res) /\
      contains.copyExcept(old -> res, elem === x)
  }

  postcondition of remove {
    (old: OldState, res: NewState) =>
      not (contains(x, res)) /\
      contains.copyExcept(old -> res, elem === x)
  }

  invariant on add remove {
    (_: OldState, res: NewState) =>
      not contains(x, res)
  }
}
```

Building Geo-Distributed Apps, the ECRO Way

ECRO Data Type

Sequential implementation

```scala
case class AWSet[V](set: Set[V]) {
  def add(x: V) =
    new AWSet(set + x)

  def remove(x: V) =
    new AWSet(set - x)

  def contains(x: V) =
    set.contains(x)
}
```

DSL for distributed specification

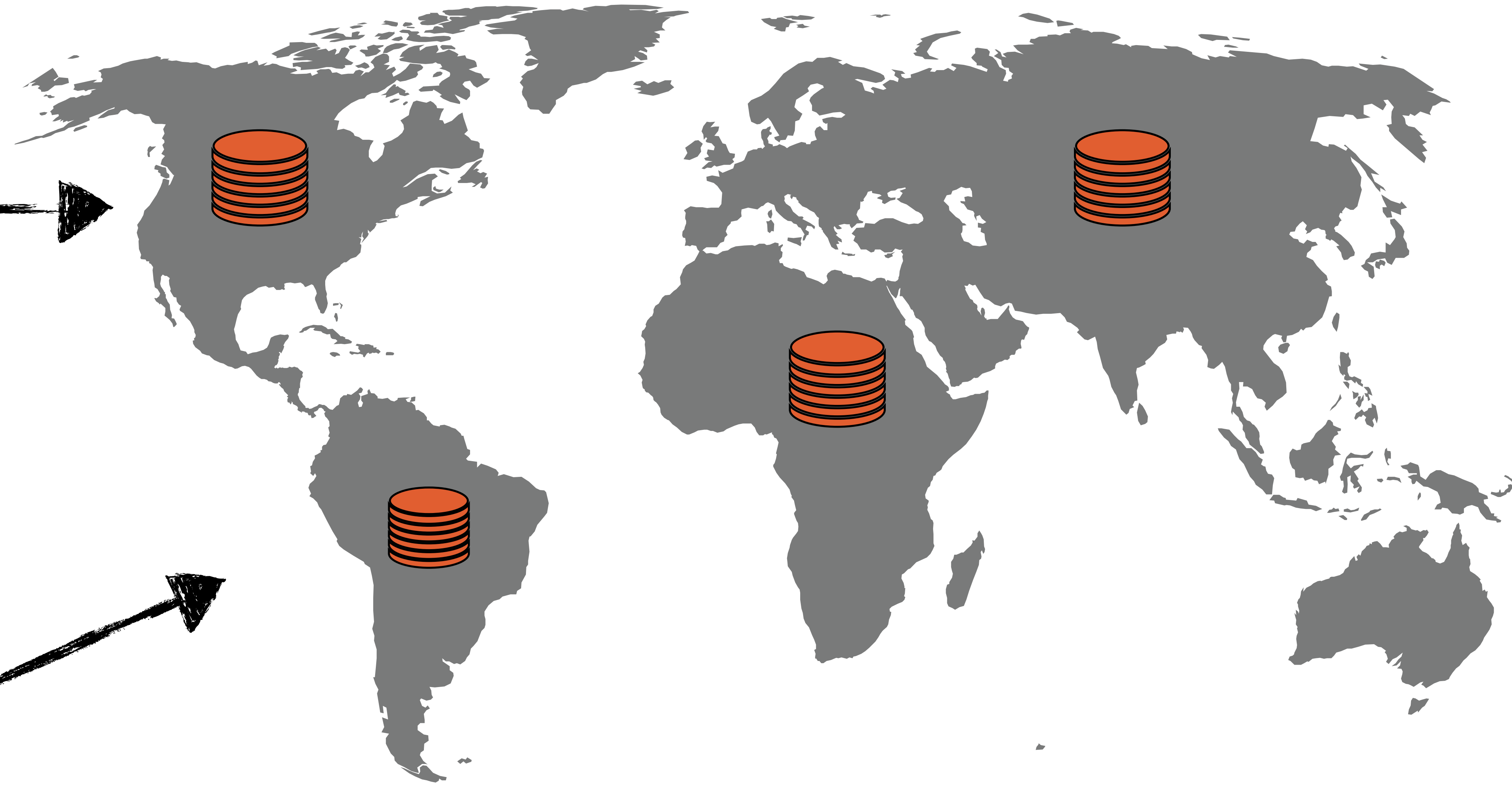
```scala
object AWSet {
  // contains: V × State × Bool
  val contains: Relation = ...

  postcondition of add {
    (old: OldState, res: NewState) =>
      contains(x, res) /\
      contains.copyExcept(old -> res, elem === x)
  }

  postcondition of remove {
    (old: OldState, res: NewState) =>
      not (contains(x, res)) /\
      contains.copyExcept(old -> res, elem === x)
  }

  invariant on add {
    (_: OldState, res: NewState) =>
      contains(x, res)
  }
}
```

Translation

Ordana Analysis Tool

SMT Solver

Replicated Data Type



Ordana: Statically Analyzes Distributed Specs

Derives information about:

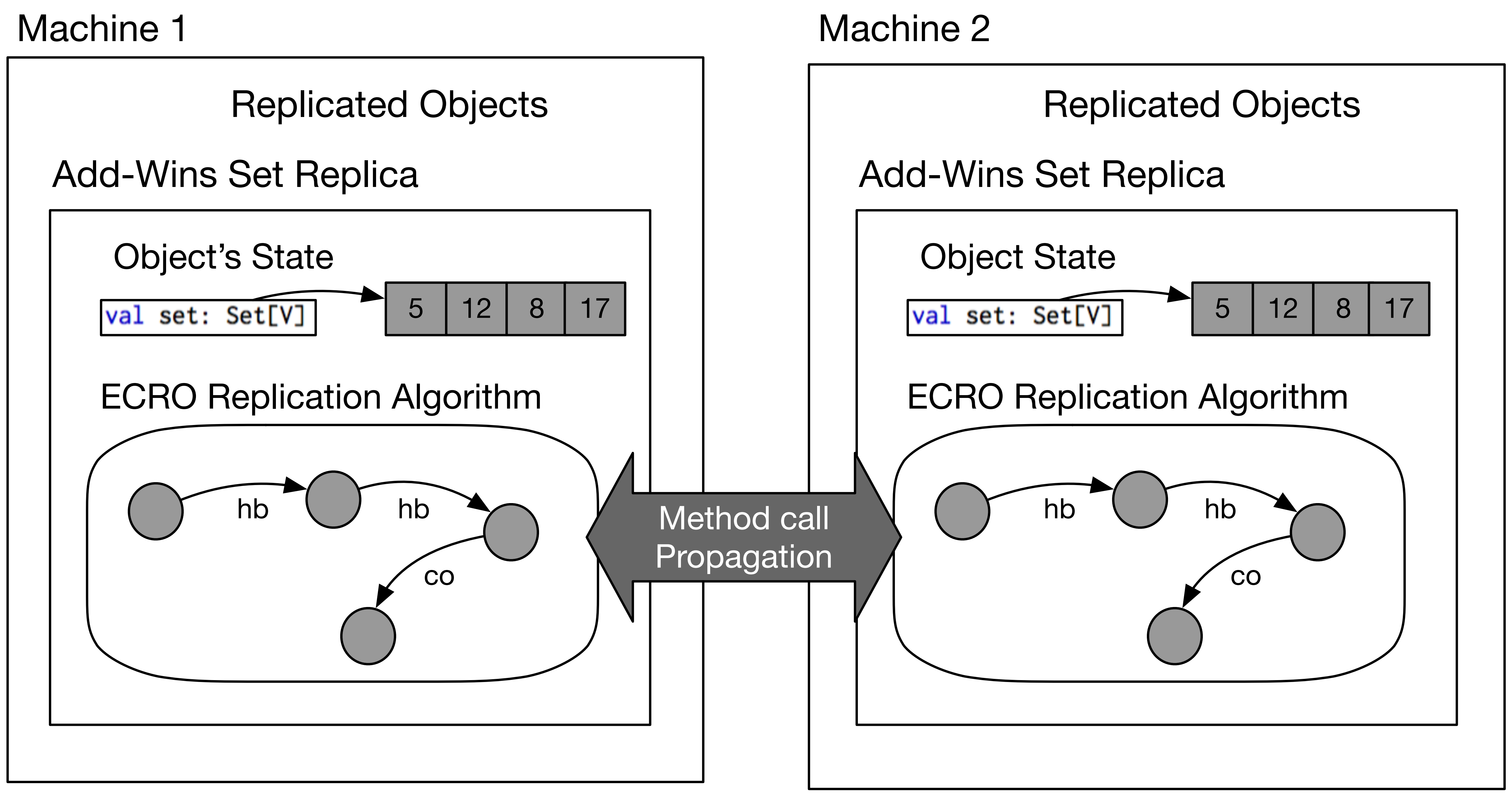
1. Commutative methods
2. Conflicting methods

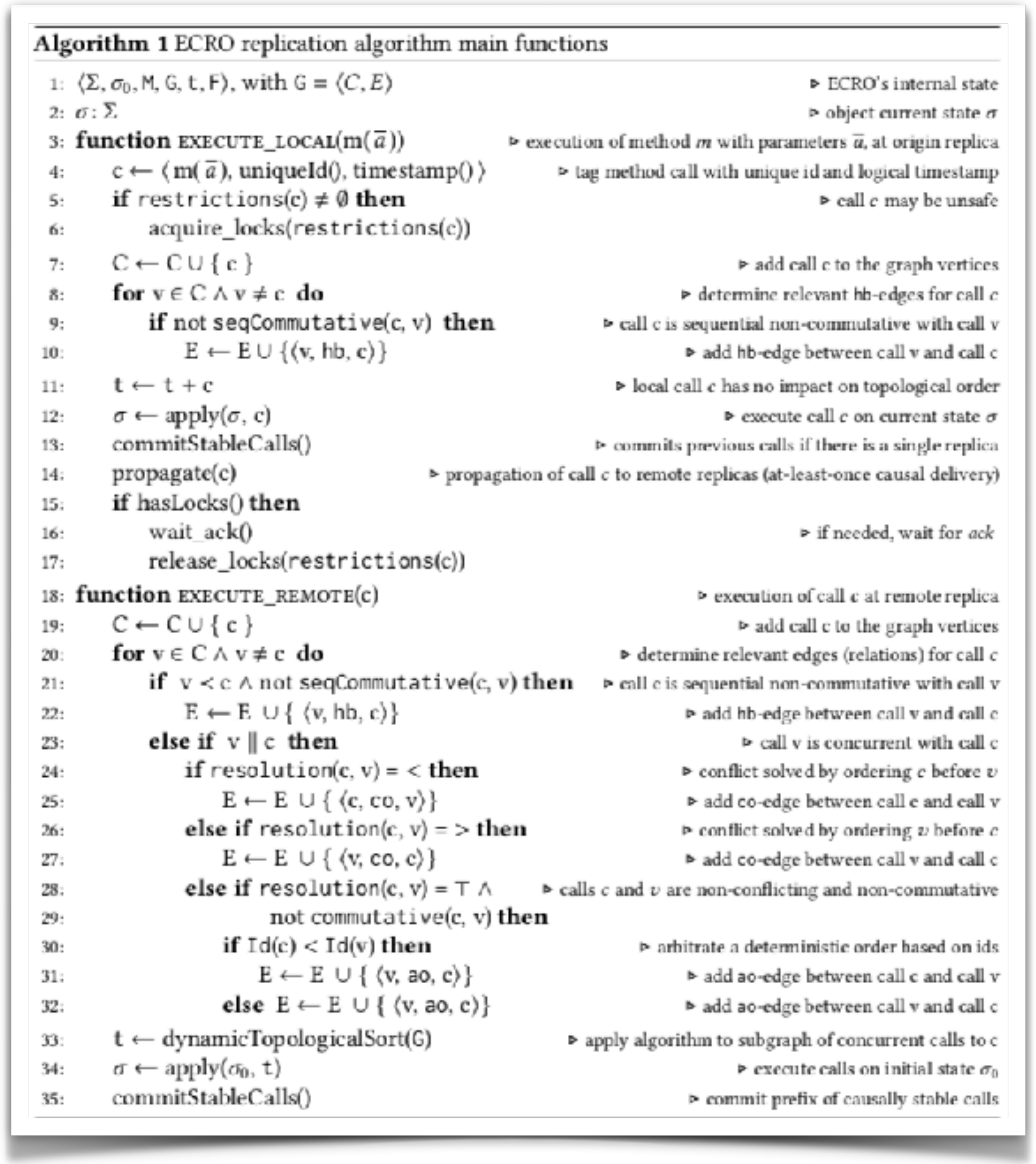
And finds:

3. Coordination-free solutions to conflicts
4. Fine-grained locks if no solution can be found

Serializing Operations: the ECRO Algorithm

- Replicas serialize operations locally
    - strong *convergence*
    - invariant preservation (i.e. *safety*)





Algorithm 1 ECRO replication algorithm main functions

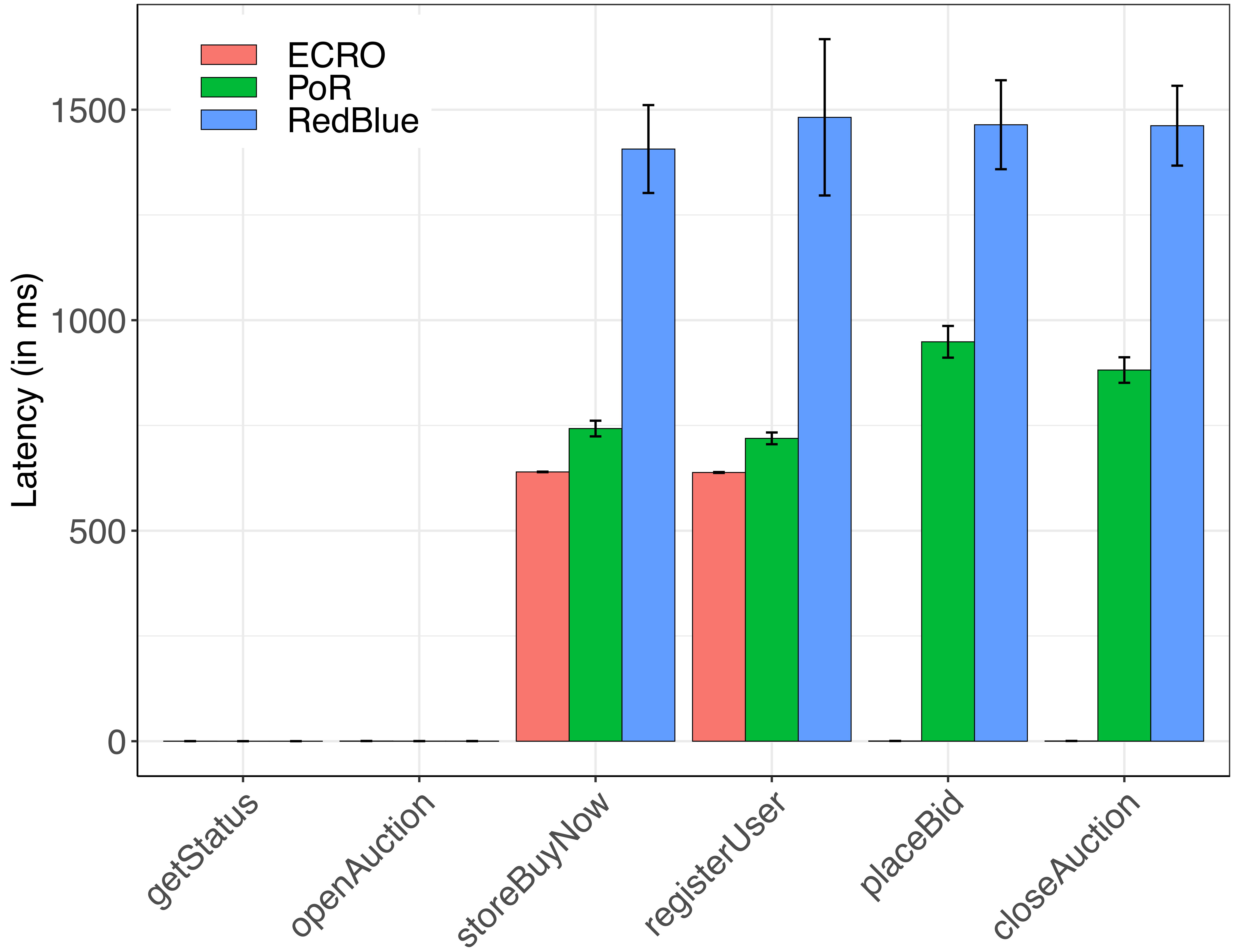
```
⟨Σ, σ0, M, G, t, F⟩, with G = ⟨C, E⟩                          ▷ ECRO's internal state
σ : Σ                                                          ▷ object current state σ
function EXECUTE_LOCAL(m(ā))        ▷ execution of method m with parameters ā, at origin replica
    c ← ⟨ m(ā), uniqueId(), timestamp() ⟩   ▷ tag method call with unique id and logical timestamp
    if restrictions(c) ≠ ∅ then                                 ▷ call c may be unsafe
        acquire_locks(restrictions(c))
    C ← C ∪ { c }                                              ▷ add call c to the graph vertices
    for v ∈ C ∧ v ≠ c do                                       ▷ determine relevant hb-edges for call c
        if not seqCommutative(c, v) then            ▷ call c is sequential non-commutative with call v
            E ← E ∪ {⟨v, hb, c⟩}                               ▷ add hb-edge between call v and call c
    t ← t + c                                       ▷ local call c has no impact on topological order
    σ ← apply(σ, c)                                            ▷ execute call c on current state σ
    commitStableCalls()                             ▷ commits previous calls if there is a single replica
    propagate(c)                    ▷ propagation of call c to remote replicas (at-least-once causal delivery)
    if hasLocks() then
        wait_ack()                                             ▷ if needed, wait for ack
        release_locks(restrictions(c))
function EXECUTE_REMOTE(c)                                     ▷ execution of call c at remote replica
    C ← C ∪ { c }                                              ▷ add call c to the graph vertices
    for v ∈ C ∧ v ≠ c do                          ▷ determine relevant edges (relations) for call c
        if v < c ∧ not seqCommutative(c, v) then    ▷ call c is sequential non-commutative with call v
            E ← E ∪ { ⟨v, hb, c⟩}                              ▷ add hb-edge between call v and call c
        else if v ∥ c then                                     ▷ call v is concurrent with call c
            if resolution(c, v) = < then                       ▷ conflict solved by ordering c before v
                E ← E ∪ { ⟨c, co, v⟩}                          ▷ add co-edge between call c and call v
            else if resolution(c, v) = > then                  ▷ conflict solved by ordering v before c
                E ← E ∪ { ⟨v, co, c⟩}                          ▷ add co-edge between call v and call c
            else if resolution(c, v) = ⊤ ∧         ▷ calls c and v are non-conflicting and non-commutative
                    not commutative(c, v) then
                if Id(c) < Id(v) then                          ▷ arbitrate a deterministic order based on ids
                    E ← E ∪ { ⟨v, ao, c⟩}                      ▷ add ao-edge between call c and call v
                else E ← E ∪ { ⟨v, ao, c⟩}                     ▷ add ao-edge between call v and call c
    t ← dynamicTopologicalSort(G)              ▷ apply algorithm to subgraph of concurrent calls to c
    σ ← apply(σ0, t)                                           ▷ execute calls on initial state σ0
    commitStableCalls()                                        ▷ commit prefix of causally stable calls
```

Validation

Portfolio of ECROs

Performance of ECROs vs PoR and RedBlue consistency

Well-known CRDTs
Counter
EW-Flag
DW-Flag
AW-Set
RW-Set
AW-Map
RW-Map
List
No CRDT
Stack
Queue
Application Specific
RUBiS



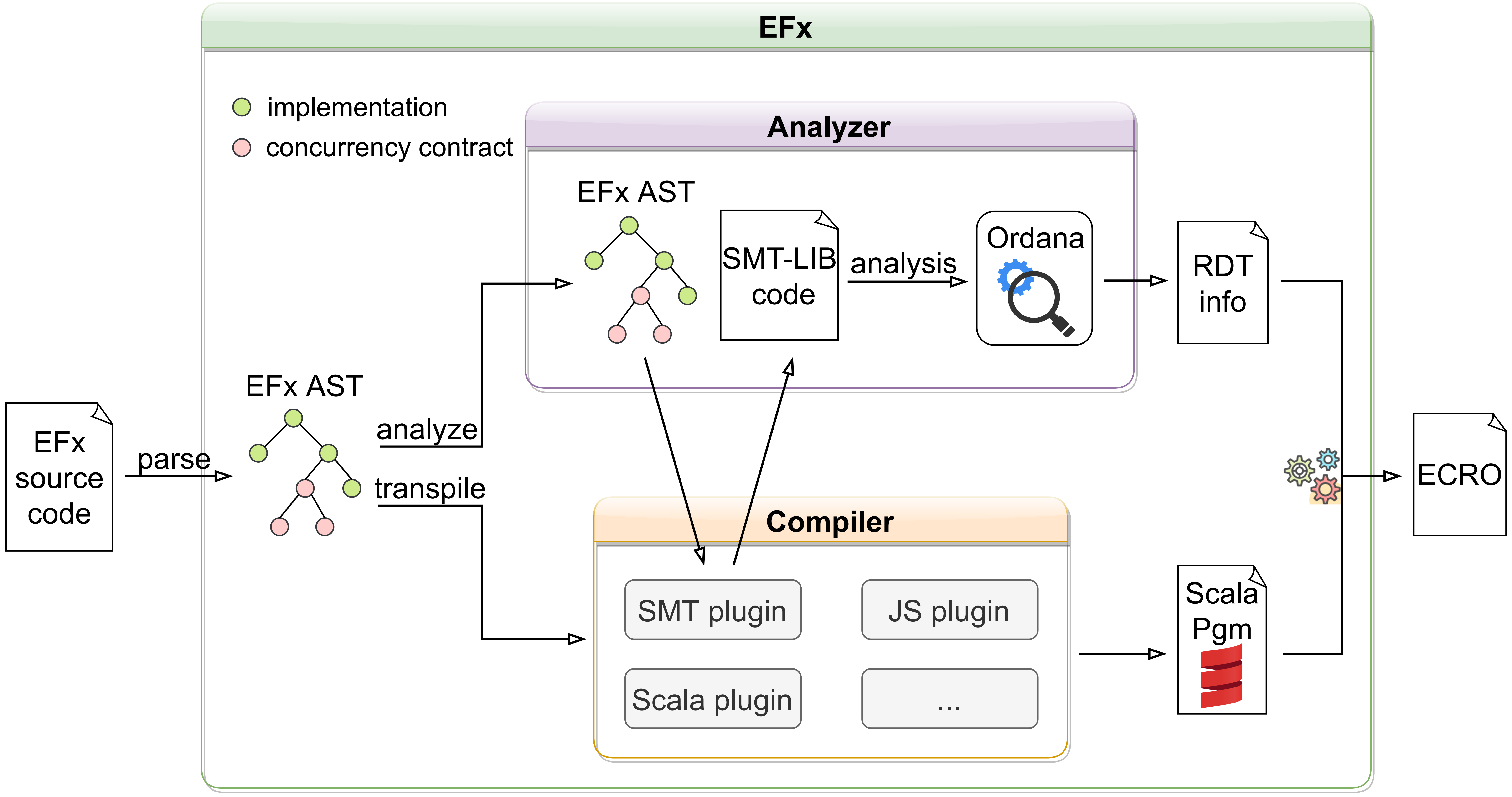
ECROs: Take Aways

- Augment sequential DT with distributed specification
- Static analysis is key to derive efficient RDTs
  - allows for informed decision at runtime
- But… separate specification
  - in FOL —> non-trivial, error-prone
  - subtle errors —> runtime anomalies
  - must evolve along with the code

How to ease the development of ECROs?

- High-level OOP language for sequential DTs 
- Define concurrency semantics and invariants 
- Fully compilable to SMT
  —> FOL specifications for free
- Synthesizes ECROs

The EFx language



Add-Wins Set in EFx


```
trait SetOps[V] {
  val set: Set[V]
  protected def copy(set: Set[V]): SetOps[V]

  def contains(elem: V) = this.set.contains(elem)
  def add(elem: V)      = this.copy(this.set.add(elem))
  def remove(elem: V)   = this.copy(this.set.remove(elem))
}
```

```
@replicated
class AWSet[V](set: Set[V]) extends SetOps[V] {
  protected def copy(set: Set[V]) =
    new AWSet(set)

  // add wins
  inv add(elem: V) {
    this.contains(elem)
  }
}
```

Remove-Wins Set in EFx


```
trait SetOps[V] {
  val set: Set[V]
  protected def copy(set: Set[V]): SetOps[V]

  def contains(elem: V) = this.set.contains(elem)
  def add(elem: V)      = this.copy(this.set.add(elem))
  def remove(elem: V)   = this.copy(this.set.remove(elem))
}
```

```
@replicated
class AWSet[V](set: Set[V]) extends SetOps[V] {
  protected def copy(set: Set[V]) =
    new AWSet(set)

  // add wins
  inv add(elem: V) {
    this.contains(elem)
  }
}
```

```
@replicated
class RWSet[V](set: Set[V]) extends SetOps[V] {
  protected def copy(set: Set[V]) =
    new RWSet(set)

  // remove wins
  inv remove(elem: V) {
    !this.contains(elem)
  }
}
```

Validation: Portfolio of RDTs

ECRO portfolio

Application specific

Data Type	LoC	C	M	Description and distributed semantics
Counter	6	1	2	Supports increments and decrements.
EW-Flag	13	1	2	Flag that can be enabled and disabled. Enable wins over concurrent disable operations.
DW-Flag	13	1	2	Similar to EW-Flag but guarantees disable-wins semantics.
AW-Set	12	1	2	Set providing add-wins semantics for concurrent adds and removes of the same element.
RW-Set	12	1	2	Set providing remove-wins semantics.
LWW-Set	11	1	2	Set providing last-writer-wins semantics.
LWW-Array	21	1	1	Array providing last-writer-wins semantics for concurrent writes on the same index.
Sync-Array	24	1	1	Array with coordinated writes (locks index before writing).
AW-Map	16	1	2	Map with add-wins semantics for concurrent adds and removes of the same key, and last-writer-wins semantics for concurrent adds of the same key.
RW-Map	16	1	2	Similar to AW-Map but remove-wins semantics for concurrent adds and removes of the same key.
Stack	14	1	2	Stack allowing push, pop, and top operations. Push operations execute optimistically and are totally ordered. Pop operations are coordinated in order not to pop more elements than there are on the stack.
Queue	12	1	2	Enqueue operations run optimistically and are totally ordered. Dequeue operations are coordinated to avoid dequeueing more elements than there are in the queue.
VotingGame	53	3	2	A distributed voting game inspired by contemporary tv-shows [Cet+14].
SmallBank	90	2	4	Banking application corresponding to the SmallBank benchmark [Alo+08].
RUBiS	87	2	6	Auction system similar to the RUBiS benchmark [EJ09].
Airline	285	9	9	An airline reservation system inspired by Acme Air [TS].

VeriFx

Correct replicated data types for the masses

Kevin De Porre, Carla Ferreira, and Elisa Gonzalez Boix. VeriFx: Correct replicated data types for the masses. In 37th European Conference on Object-Oriented Programming, ECOOP 2023, pages 9:1--9:45. Schloss Dagstuhl, July 2023.

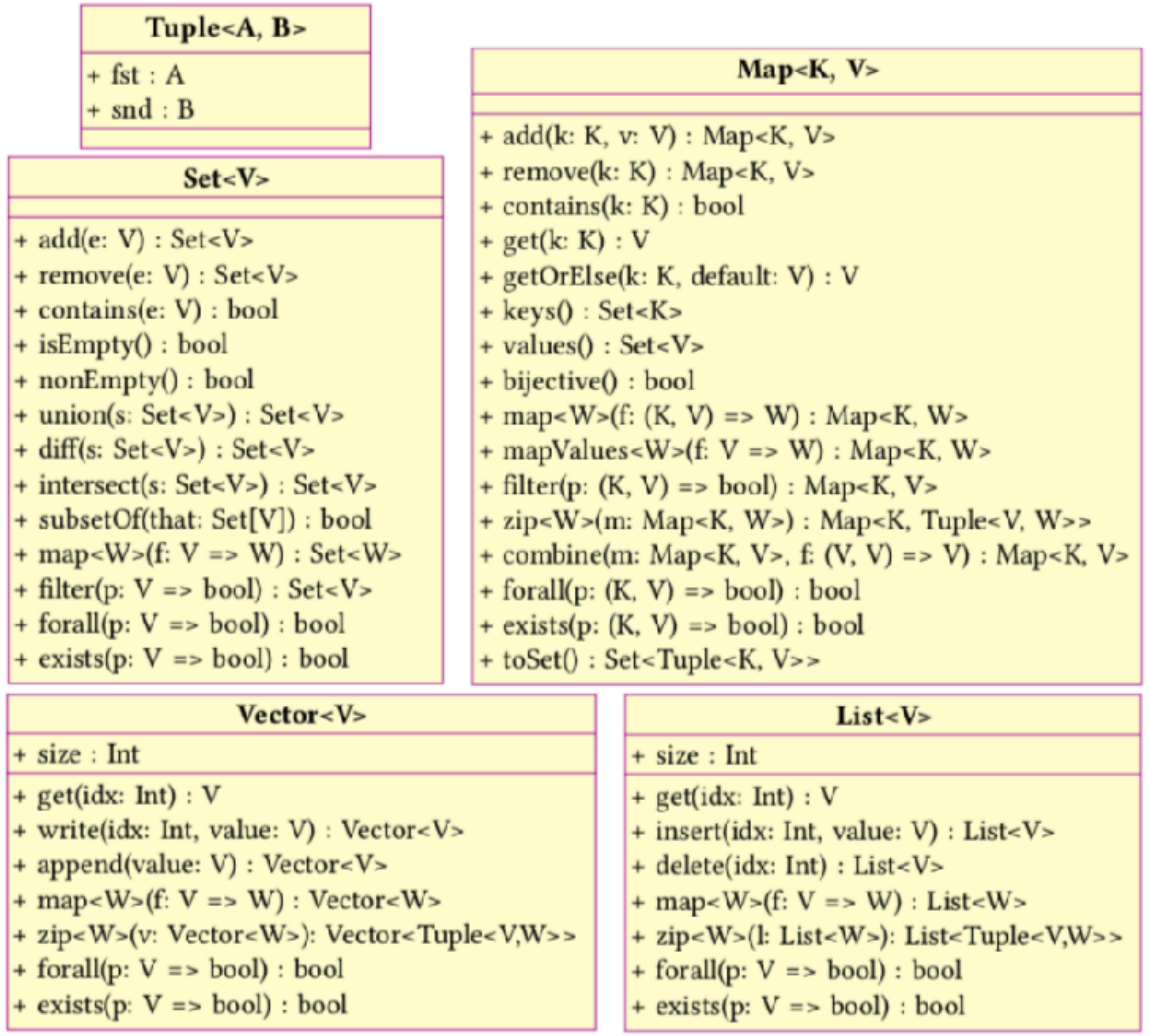
The VeriFx Language

http://verifx.org/

- High-level OOP language with extensive functional collections
  - maps, sets, vectors, etc.

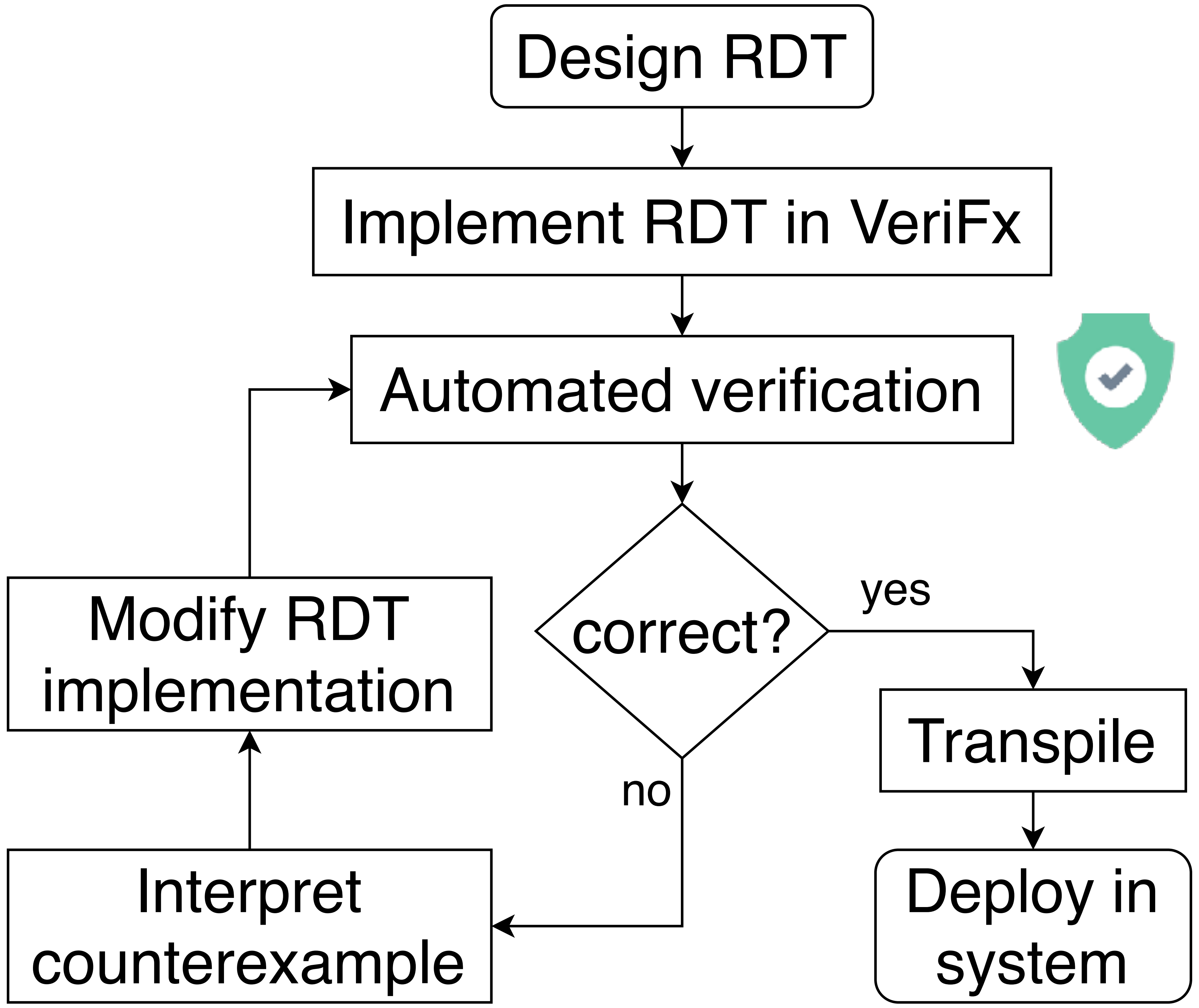
- Features a novel *proof construct*
  - used by programmers
  - describe application-specific correctness properties

- Also fully compilable to SMT
  —> Automated proof verification



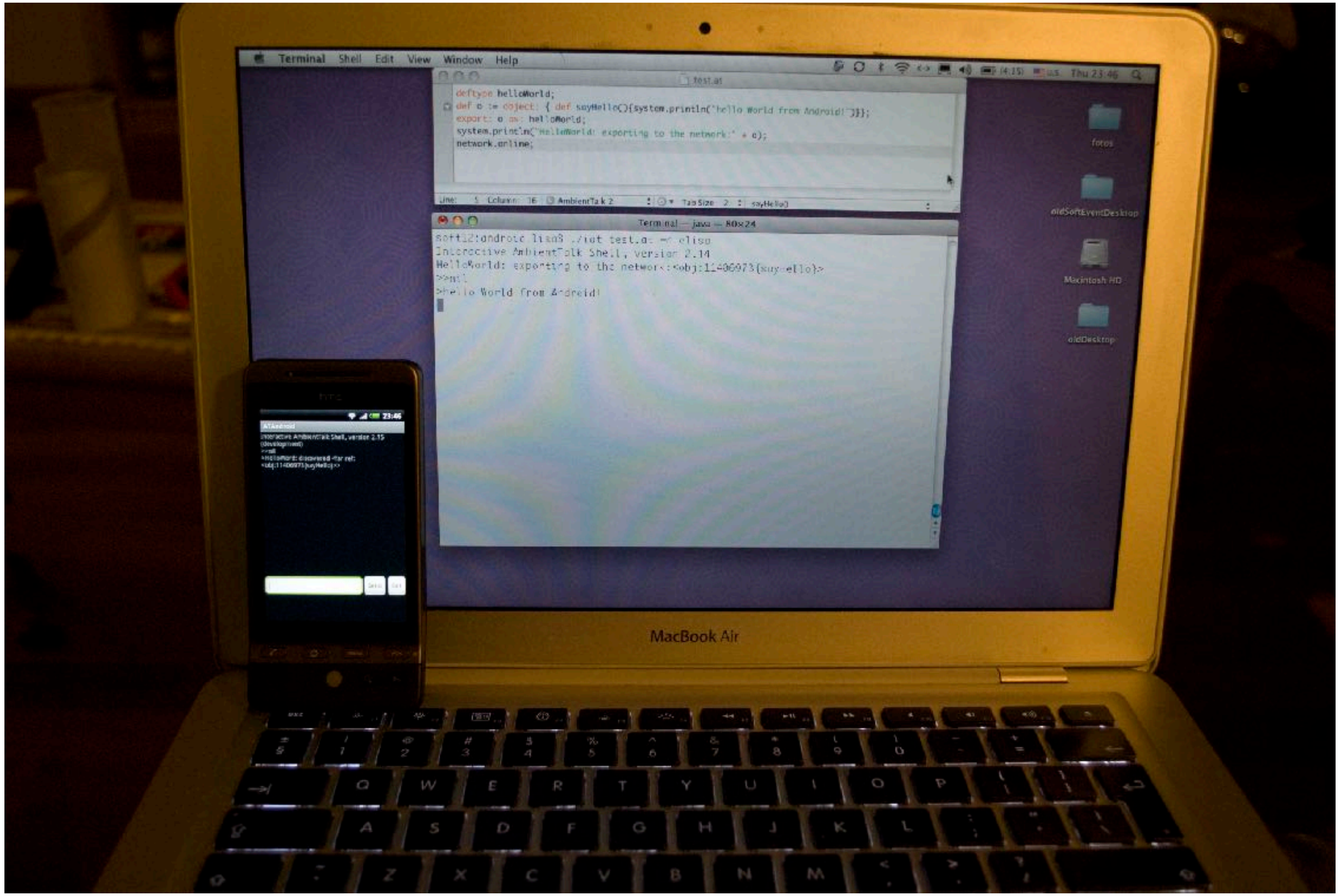
VeriFx's Iterative Workflow for developing RDTs

http://verifx.org/



Supporting development of distributed systems goes beyond providing novel programming models

Tooling is essential!



Elisa Gonzalez Boix
@elisagboix

Running around with 10000 euros for my AmbientTalk class about distributed programming on android :D

Reasoning about distributed events..

Generate and receive application requests

```
obj<-msg(arg)
def msg(param) { ... }
```

Follow-up on outstanding requests

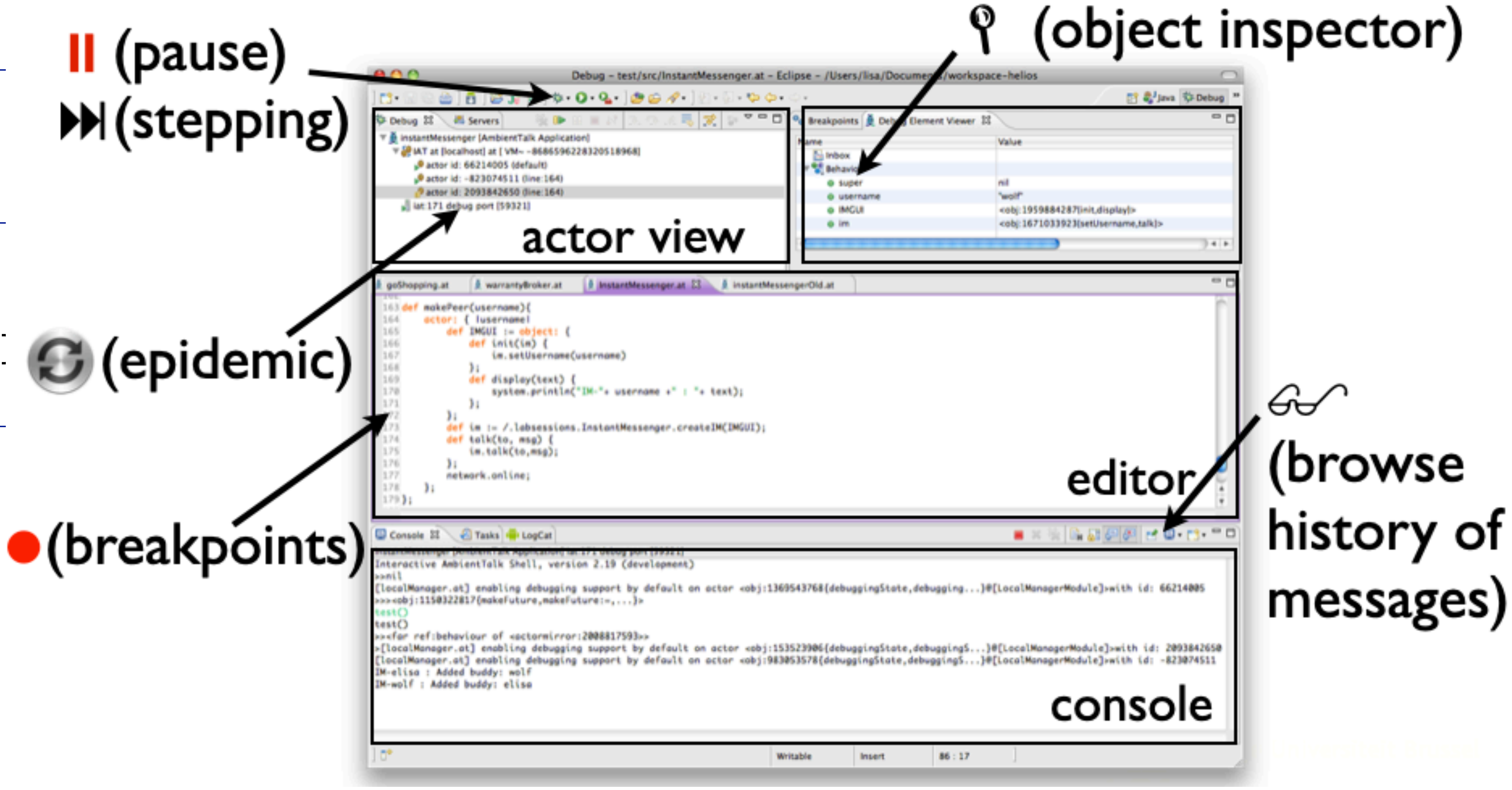
```
when: future becomes: { |result| ... }
```

React to services appearing and disappearing

```
when: type discovered: { |ref| ... }
whenever: type discovered: { |ref| ... }
```

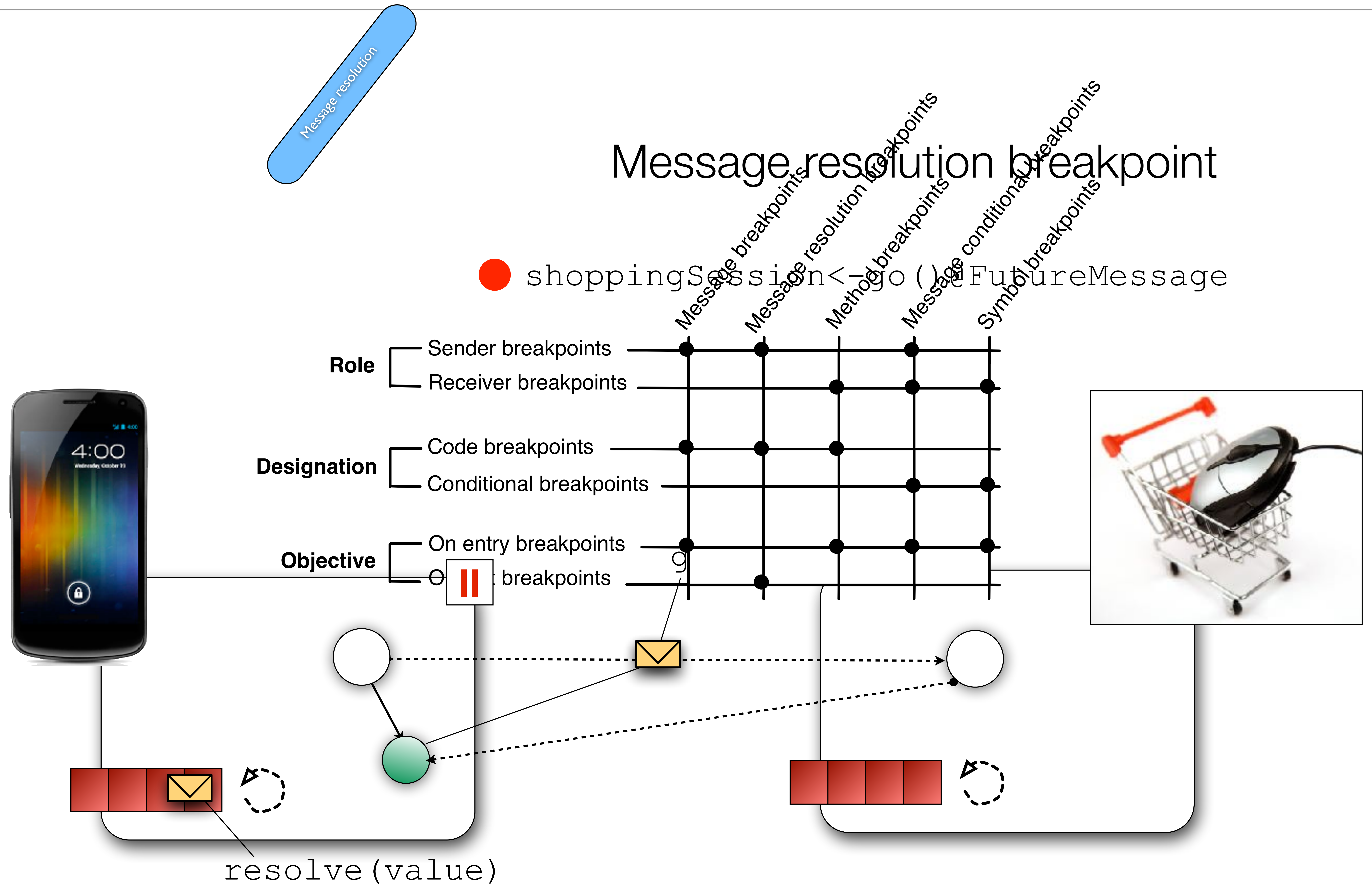
React to references disconnecting, reconnecting, and expiring

```
when: ref disconnected: { ... }
when: ref reconnected: { ... }
when: ref expired: { ... }

whenever: ref disconnected: { ... }
whenever: ref reconnected: { ... }
```



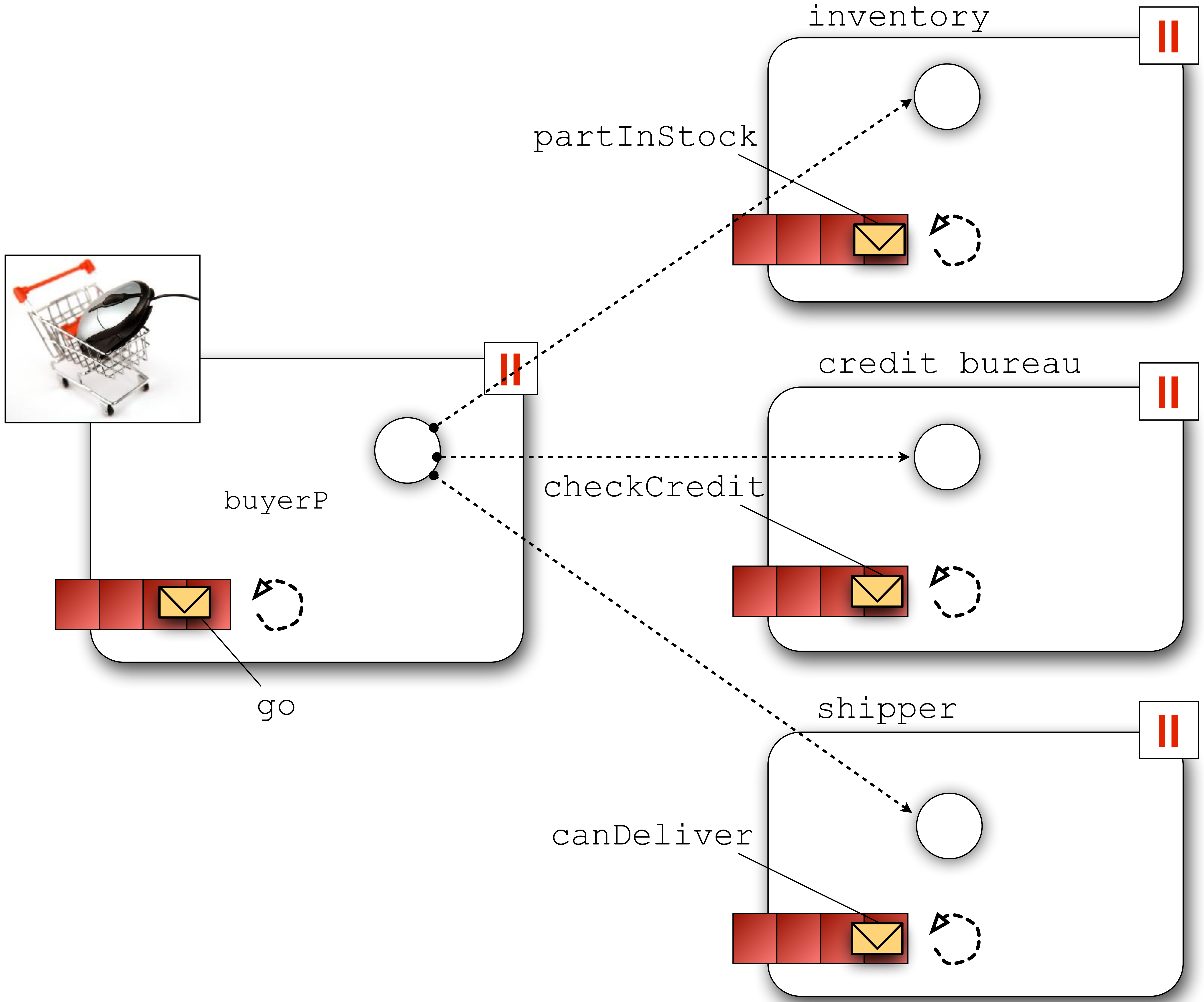
Elisa Gonzalez Boix, Carlos Noguera, and Wolfgang De Meuter. Distributed debugging for mobile networks. Journal of Systems and Software, 90:76–90, 2014.

REME-D Breakpoint Catalog



REME-D Stepping

- Step Over
- **Step Into**
- Step Return
- Step Until



Pre-experimental User Study

Goal: How real users perceive and value the features of an ambient-oriented (AmOP) debugger.

- One-group pretest-posttest quasi-experiment design.
- 22 participants.

email: `egonzale@vub.ac.be`
office: 10F731

5 goShopping: Debugging AmbientTalk programs with REME-D

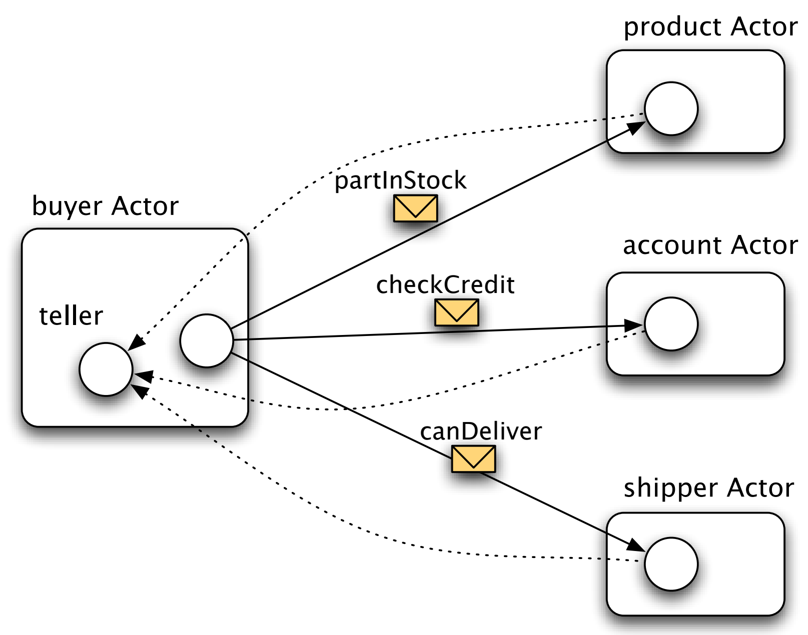
Lab session material available at Pointcarre under LabSessions, and at `http://soft.vub.ac.be/~egonzale` under Teaching.

5.1 Idea

The purpose of this exercise is to get familiar with REME-D [1], a distributed debugger designed for AmbientTalk applications. To this end, the lab material provides you with an application that contains errors. You should try to fix them by launching it in the Eclipse AmbientTalk plugin in debug mode and using REME-D's features.

5.2 Finding bugs in the goShopping application

The provided application is a sample shopping application that needs to process purchase orders. Before the shop can acknowledge the order, it must verify three things: 1) whether the requested items are still in stock, 2) whether the customer has provided valid payment information and 3) whether a shipper is available to ship the order in time. The following picture depicts this application which consists of 4 actors.

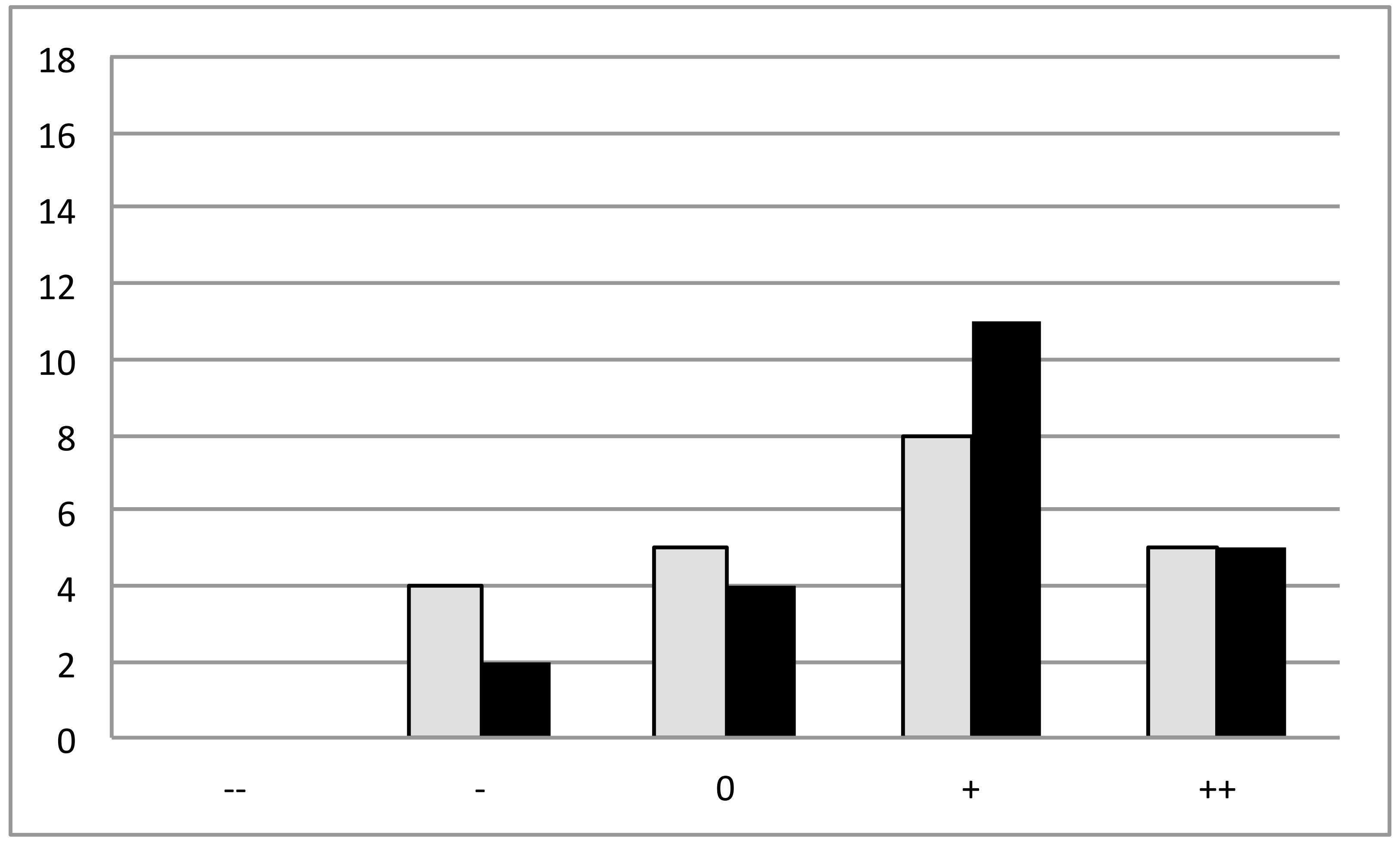


[1] read as remedy

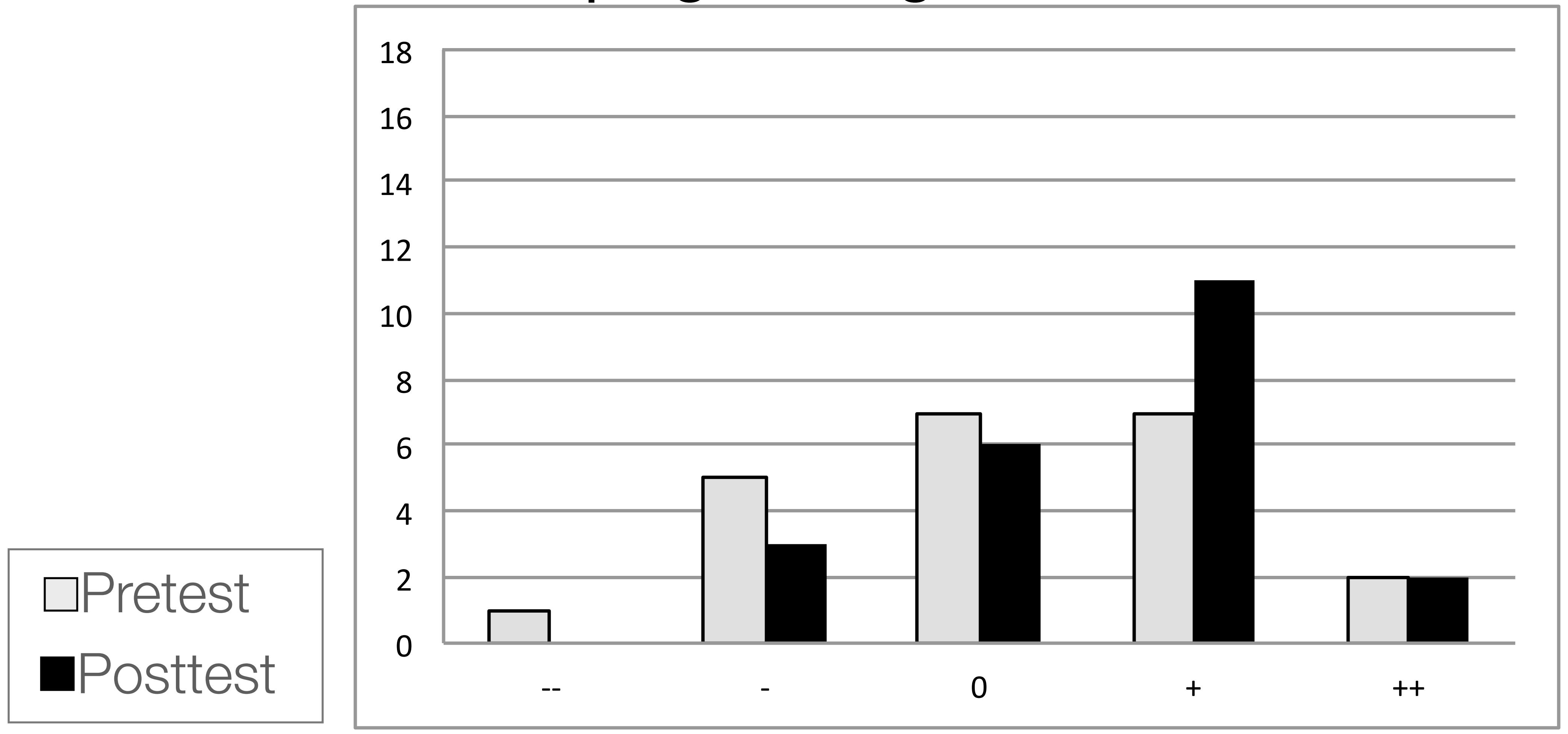
Pre-experimental User Study: Take Home Message

- Users value REME-D as tool to make AmOP programming in AmbientTalk easier.
- REME-D supports expected features for an ambient-oriented debugger.
- Impact of UI and visualisations.

Value as a tool to find bugs



Value as a tool to ease distributed programming in AmbientTalk



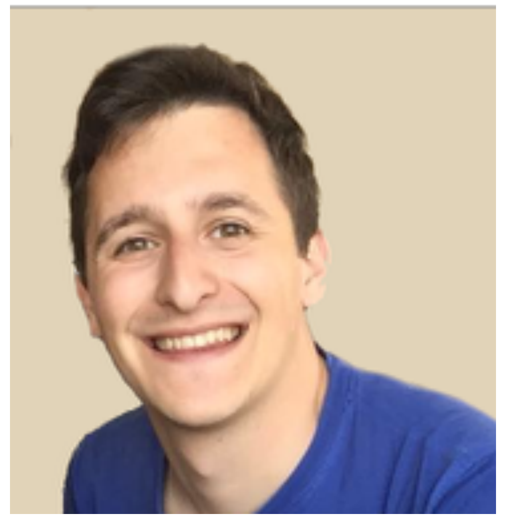
Could we build debugging support that..

DISTRIBUTION & CONCURRENCY RESEARCH GROUP

- deals with non-determinism inherent to distributed systems?
- can be applied to different concurrency models?
- features advanced visualisations for the event-based nature of distributed systems?
- is probe-effect free?
- deals with big amounts of data?
- can be used in environments with memory and network constraints?

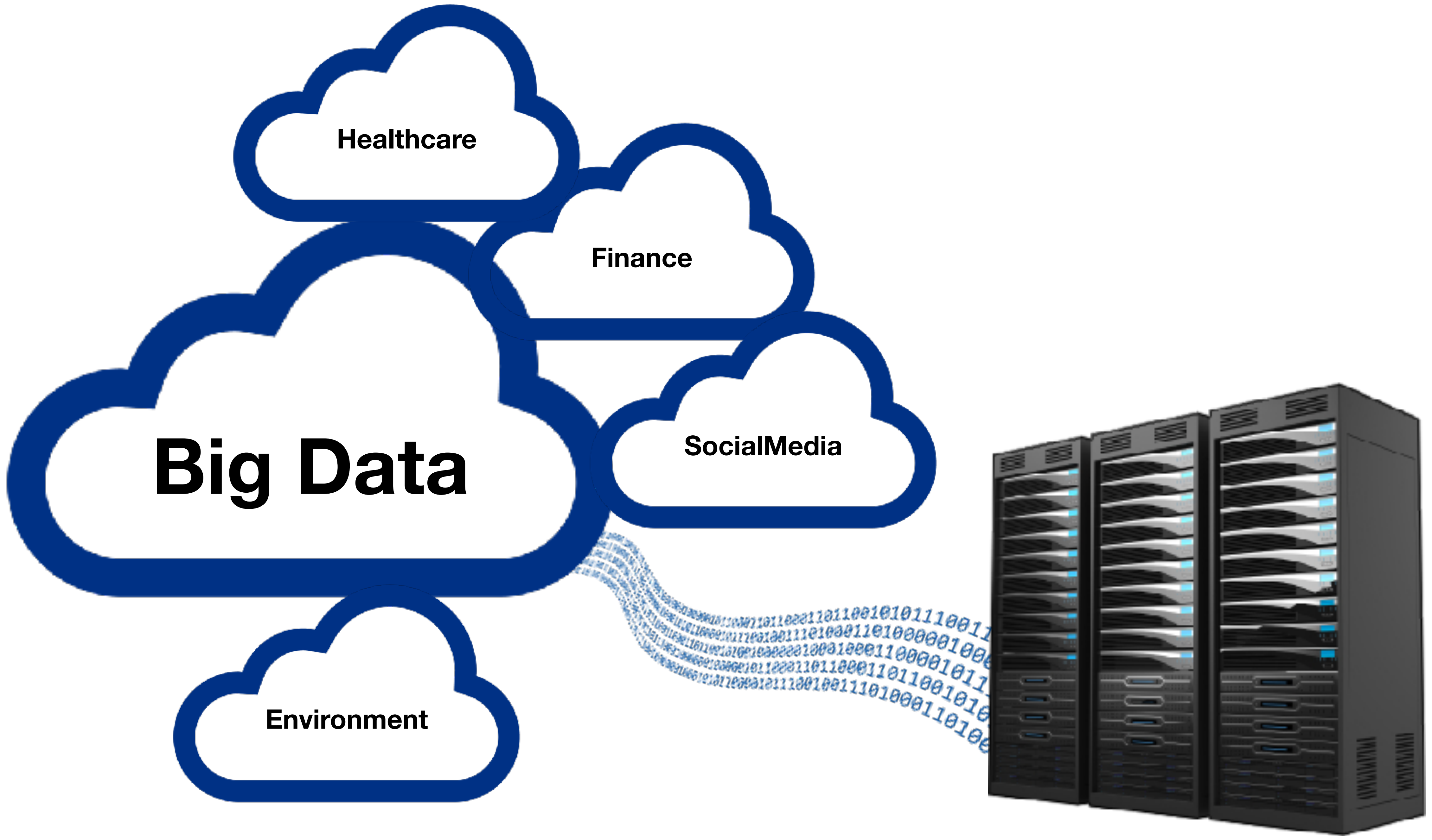
IDRA and Spa

Practical Online Debugging of Big Data Processing Applications



Matteo Marra, Guillermo Polito, and Elisa Gonzalez Boix. Practical Online Debugging of Spark-like Applications. In Proceedings of the IEEE 21st International Conference on Software Quality, Reliability and Security (QRS). IEEE, p. 620-631 12 p. 2021.

Big Data Processing



Long Running
Due to the high volume of data they have to analyze

Distributed
They remotely execute on clusters, which slows down the debugging cycle

Bugs in Big Data Processing Applications

37% of Reported Errors

In cloud Big Data processing services are attributed to developer errors [Zhou et al. 2015]

Code Defect

Explicit errors inserted by developers

Operation Fault

Common operational mistakes, e.g., file renaming

Misuse

A configuration error, e.g., using a wrong library version

Could we build a debugger so that..

Online Debugging

Debug the system when the bug happens

Avoid Replays

Global View

Centralised debugging of the distributed system

Domain-Specific Debugging

Isolation

Debug the system without interfering with its execution

Updates of the Running System

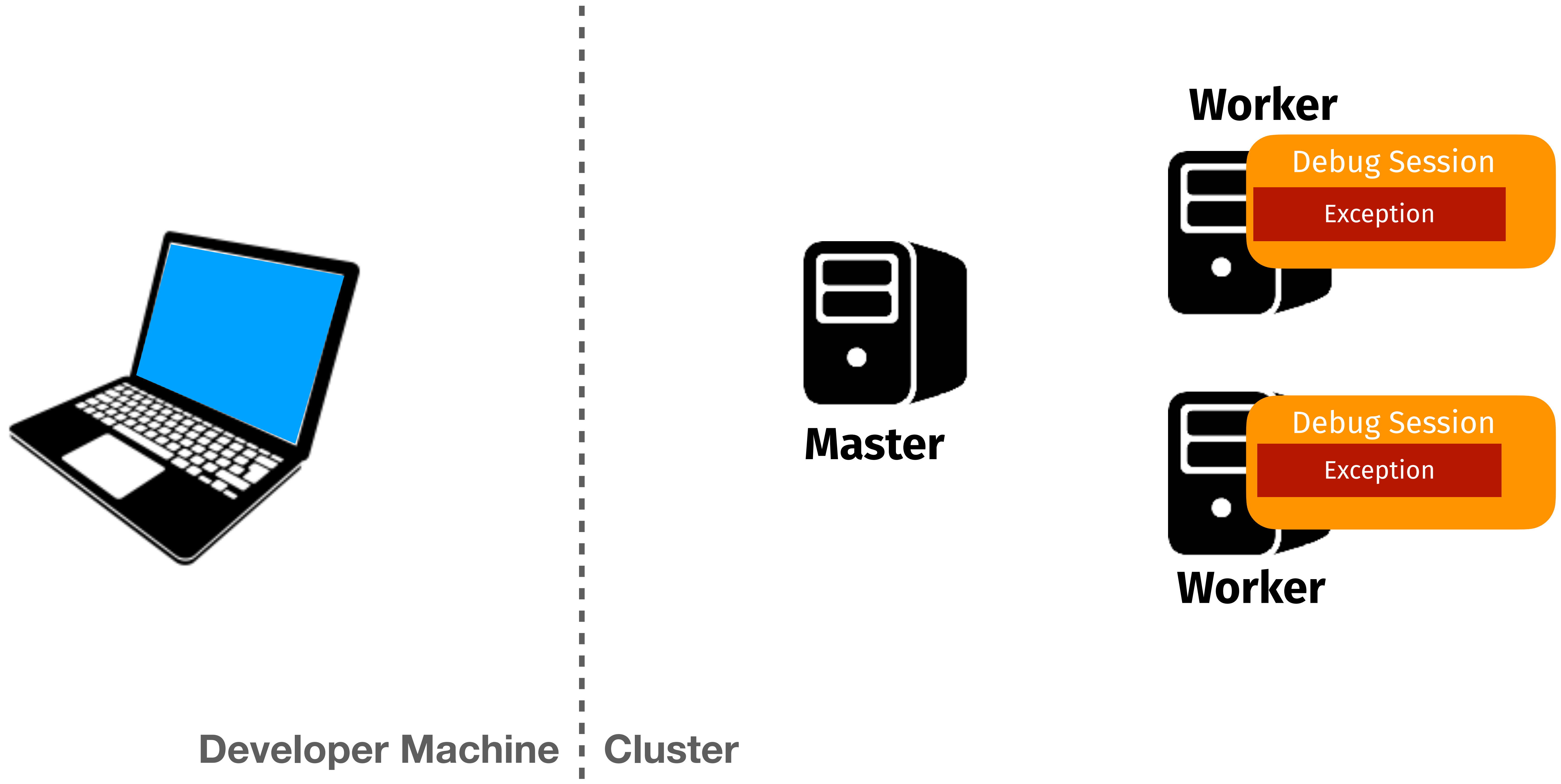
Deploy code-fixes without restarting the whole distributed system

Live Code Updating

Out-of-Place Debugging

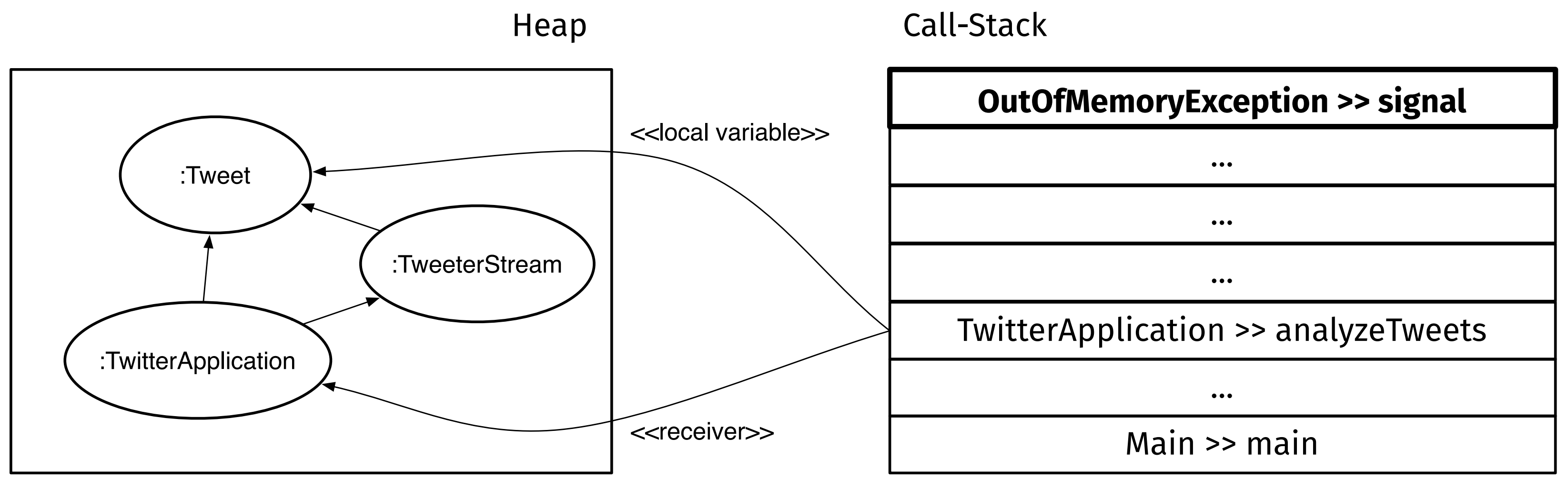
Avoid Replays

Domain-Specific Debugging



Debugging Events

Debugging Session
Captures the execution state through the call-stack



+

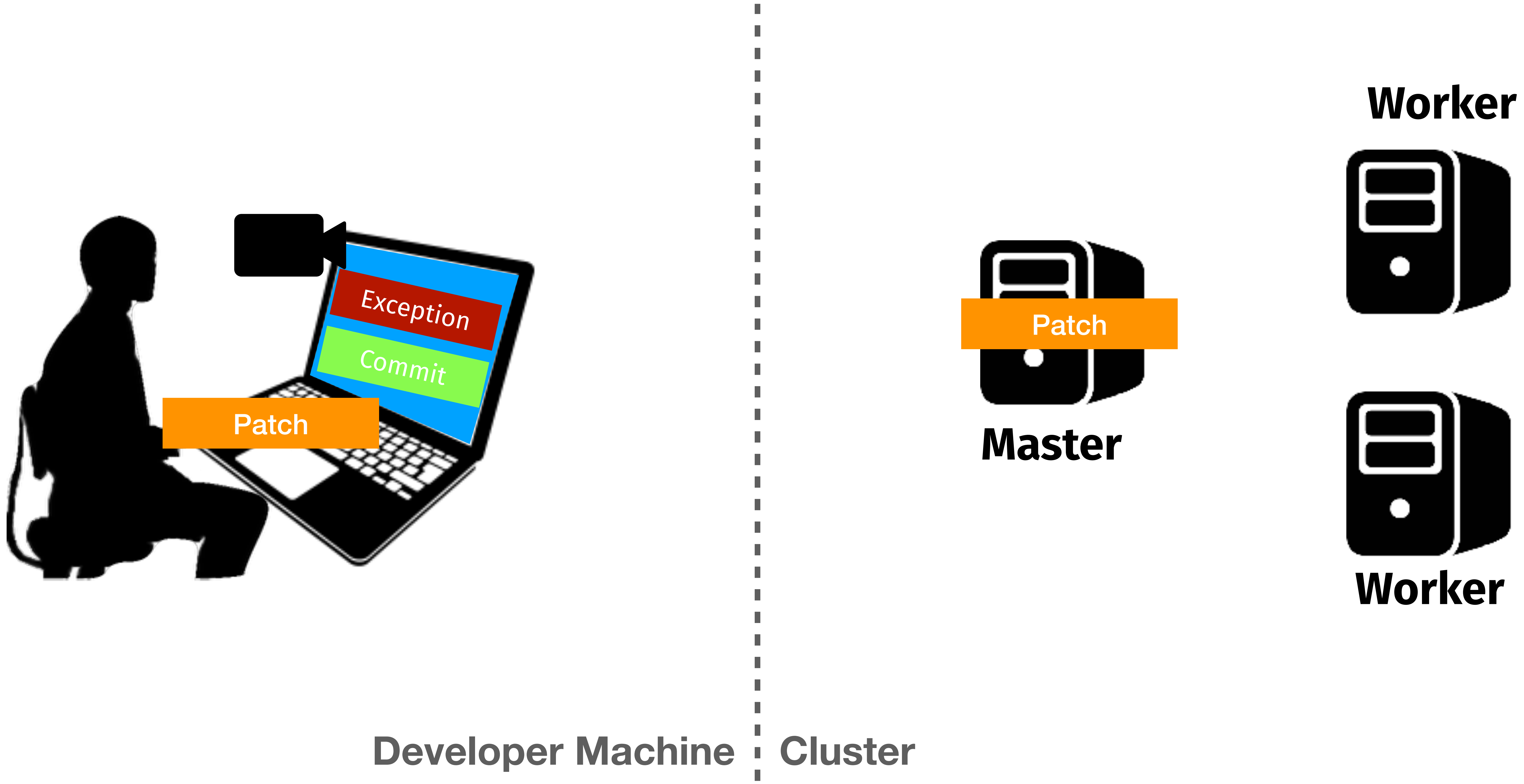
Remove Framework Frames
Reduce the amount of data to be transferred

Include the *event-inducing record*
I.e., the record that was being processed when the debugging event (breakpoint or error) happened.

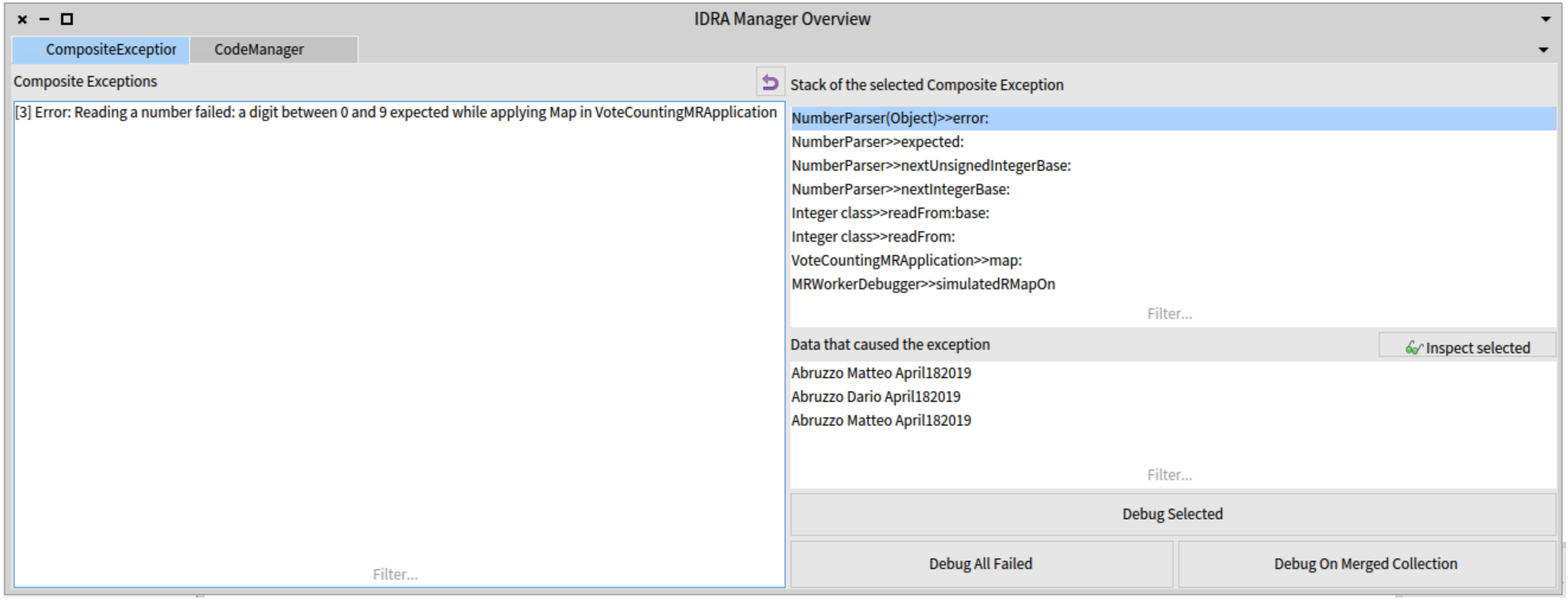
Include the partition of the *event-inducing record*
The partition of data that was being processed when the debugging event happened, that includes the *event-inducing record*

Distributed Live Code Updates

Live Code Updating



IDRA$_{MR}$: A Live Debugger for Map/Reduce



Debug Single Record

Select which debugging event to debug starting from the event-inducing record, including its partition

Debug on Virtual Partitions

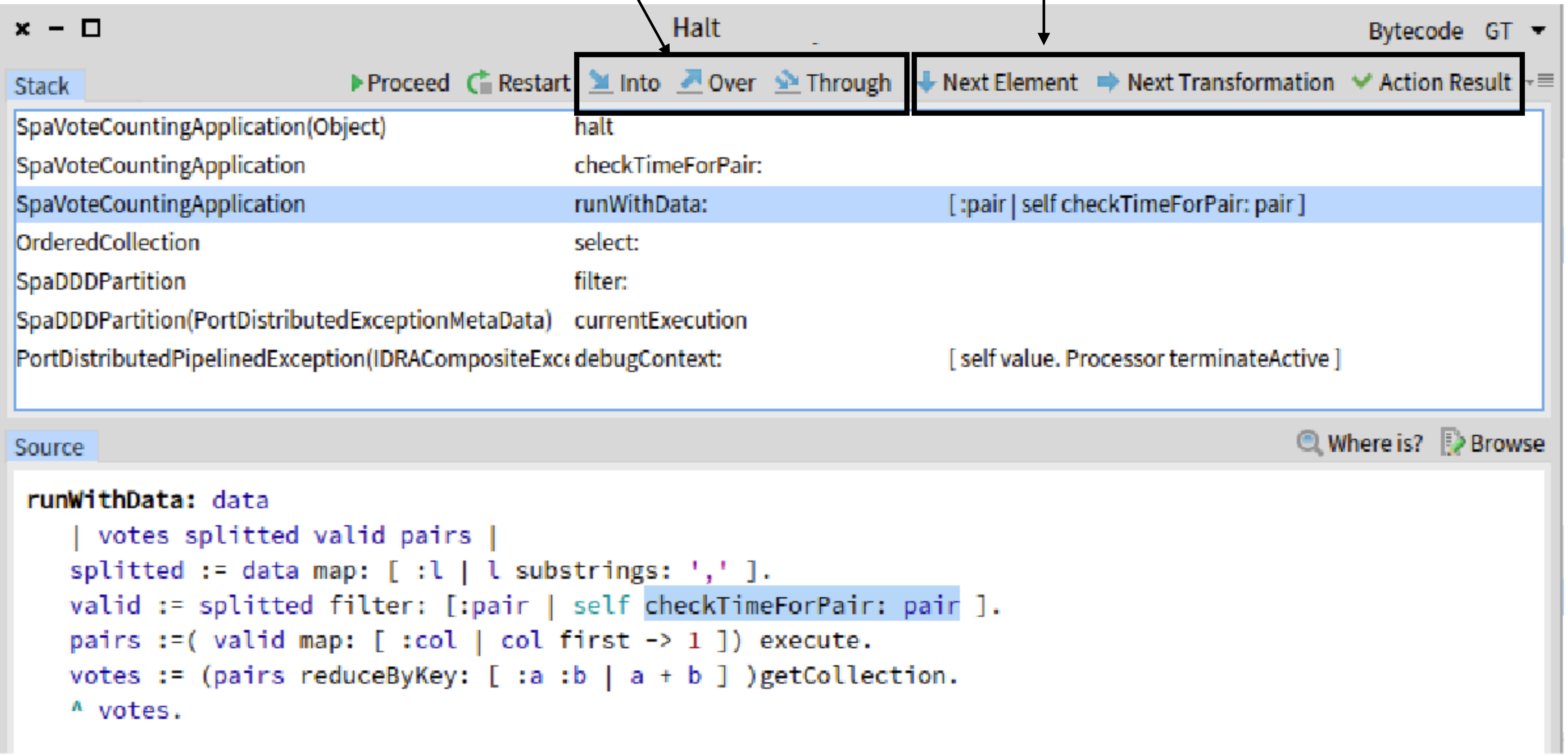
Including all of the event-inducing records, or a merge of all their partitions

Spa: a Live Debugger for Spark

Domain-Specific Debugging

Classic stepping operations
Typical of online debuggers

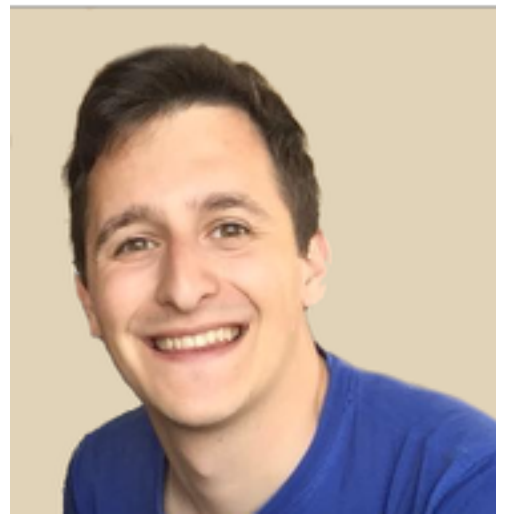
Dedicated stepping operations
Tailored to Spark-like computations



https://www.youtube.com/watch?v=GpipdhVxYq0

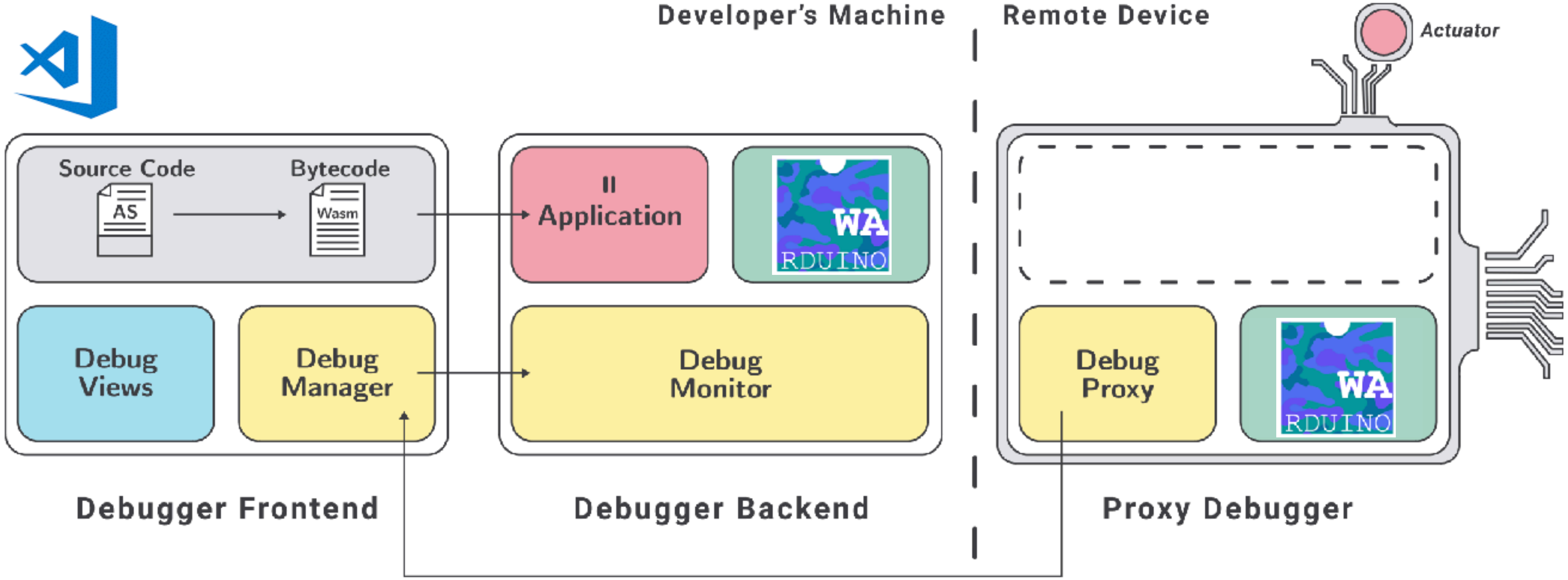
Event-based Out-of-place Debugging

Practical Online Debugging of Internet of Things applications

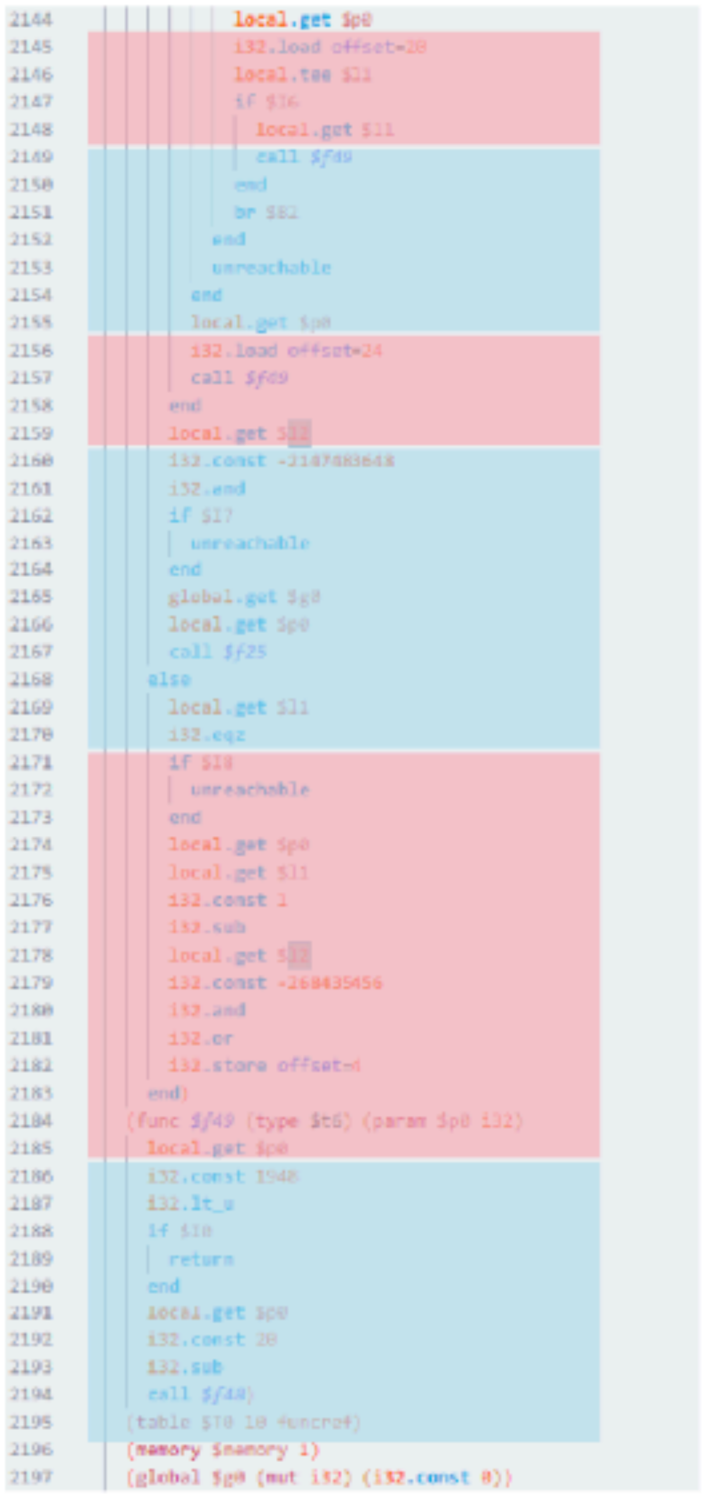

Tom Lauwaerts, Carlos Rojas Castillo, Robert Gurdeep Singh, Matteo Marra, Christophe Scholliers, and Elisa Gonzalez Boix, In, Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes (MPLR) Association for Computing Machinery (ACM), p. 85-97 13 p. 2022.

Out of Place Debugging for Internet of Things



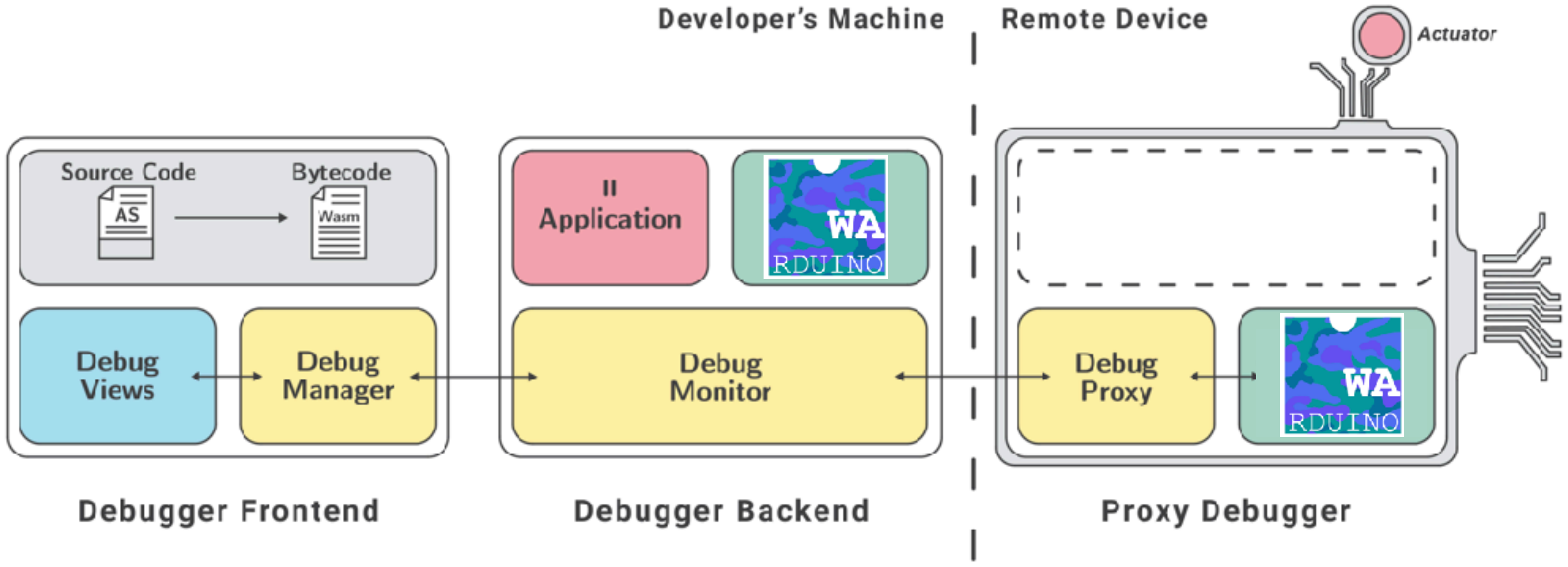
Non-transferable resources

Location Independent



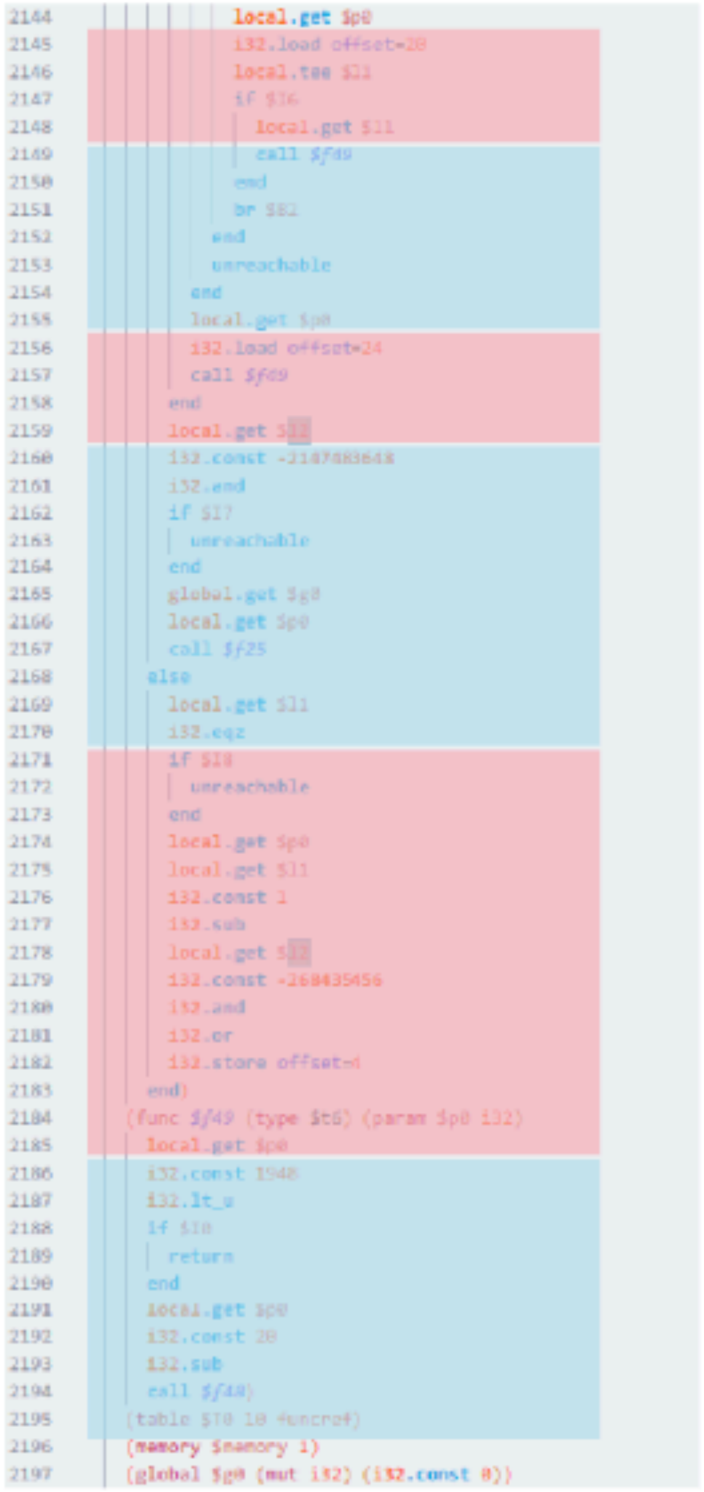
Non-transferable Resources

Pull-based



Non-transferable resources

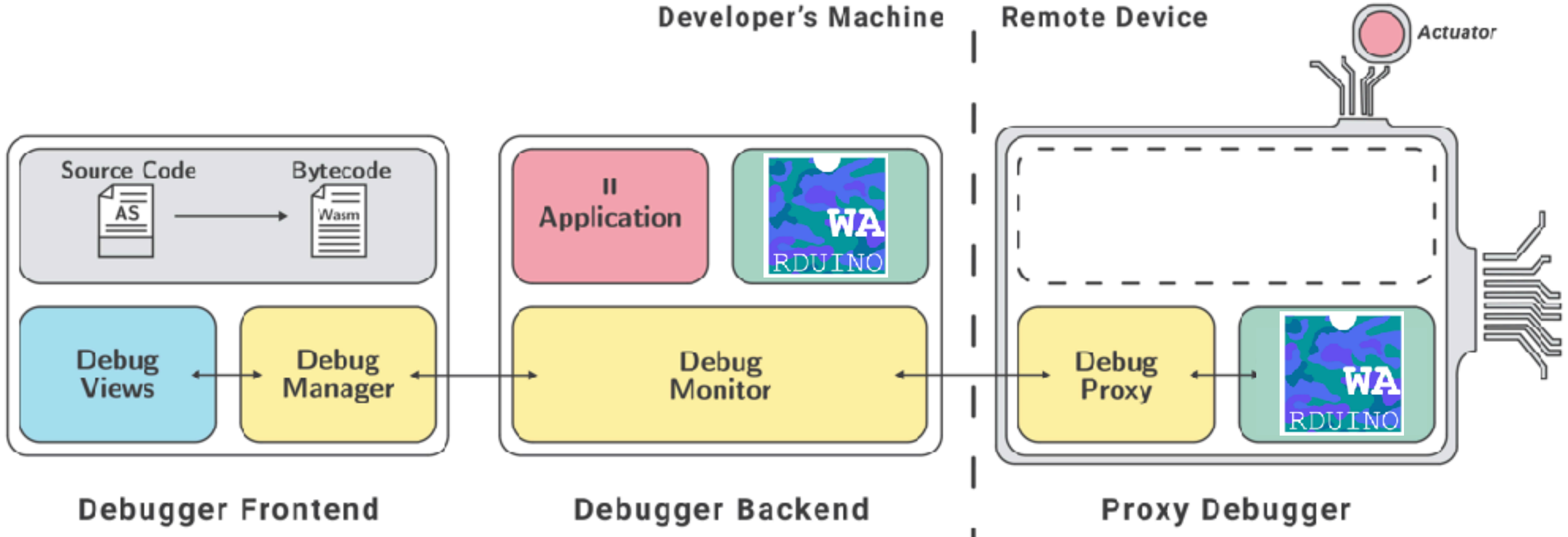
Location Independent

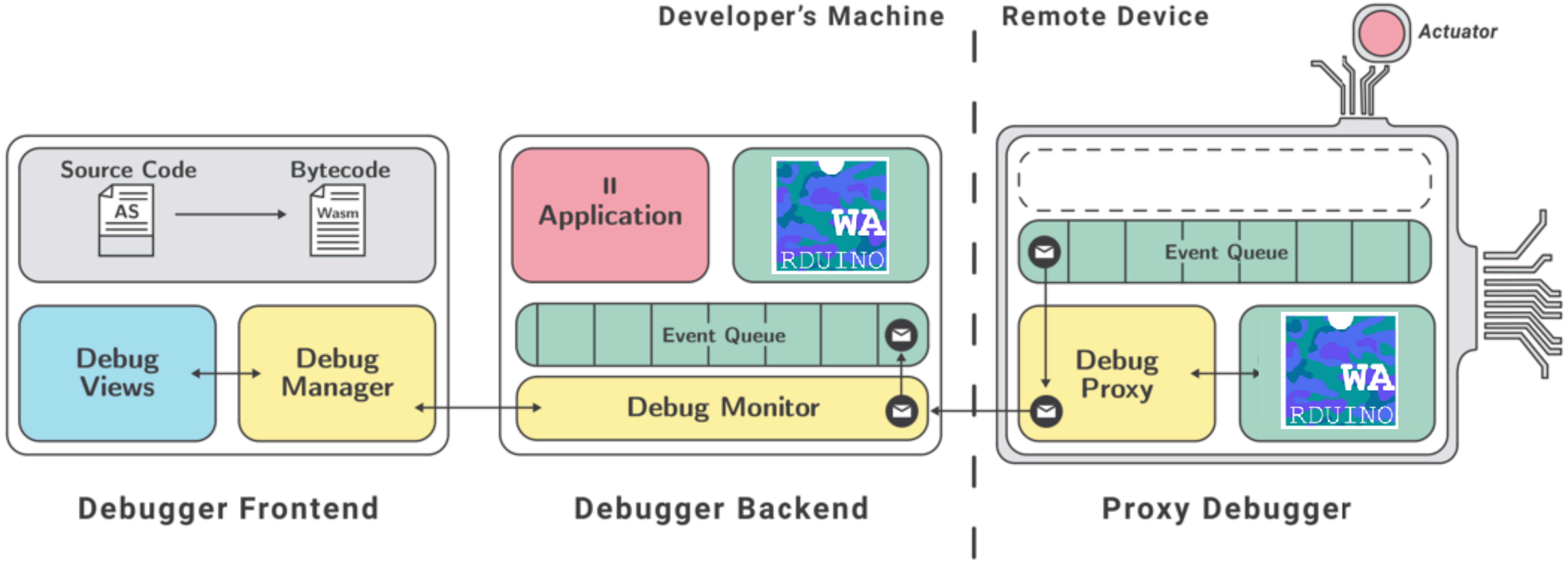


Non-transferable Resources

Pull-based

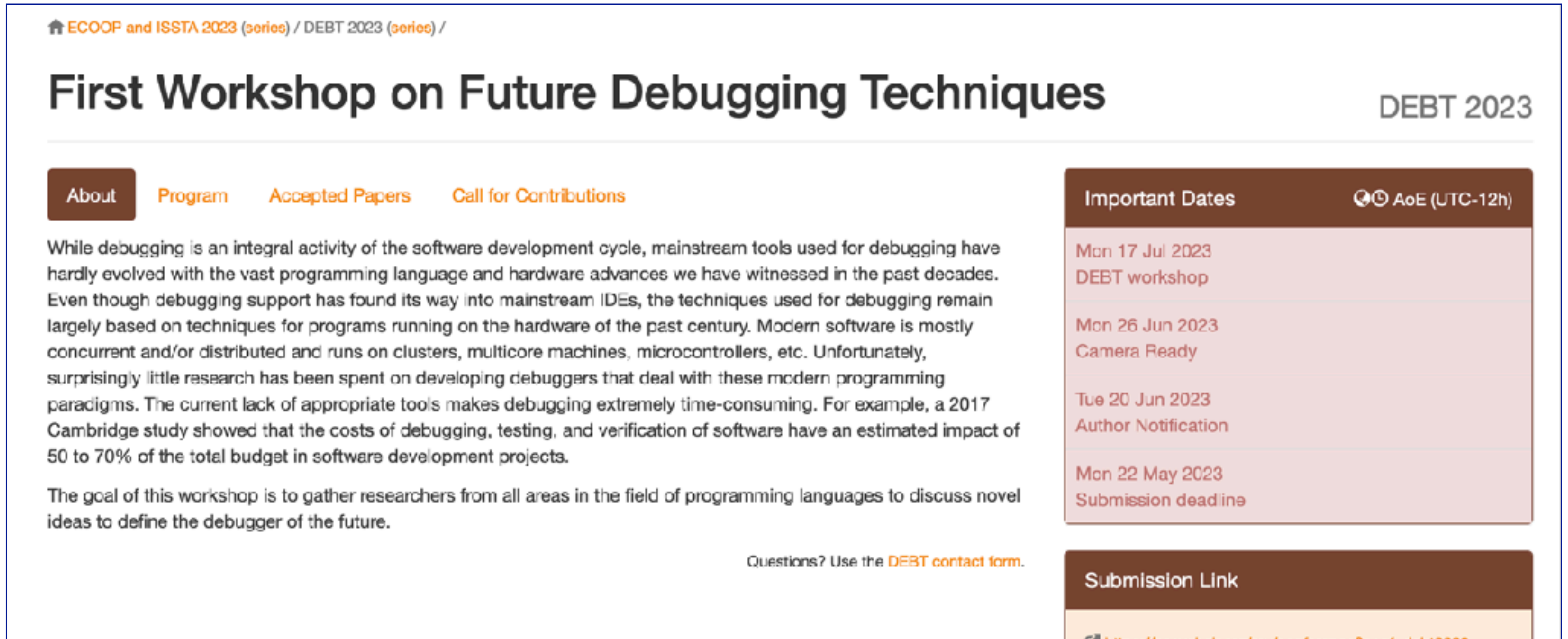
Push-based





In Conclusion

- Distributed systems are varied, successful and widespread.
- They are still challenging to design and implement.
- It is essential to explore novel programming abstractions in tandem with software tools tailored to modern concurrent and distributed software.



Thanks to DisCo & collaborators!

Dominik Aumayr	Aäron Munsters
Jim Bauwens	Florian Myter
Clément Béra	Isaac Nyabisa Oteyo
Dina Borrego	Guillermo Polito
Kevin De Porre	Nuno Preguiça
Carla Ferreira	Carlos Rojas Castillo
Robert Gurdeep Singh	Christophe Scholliers
Tom Lauwaerts	Angel Luis Scull Pupo
Stefan Marr	Carmen Torres Lopez
Matteo Marra	…
Hanspeter Mössenböck	

egonzale@vub.be
@elisagboix