Rx-CML: Migrating MultiMLton to the Cloud

KC Sivaramakrishnan

Lukasz Ziarek
SUNY Buffalo

Suresh Jagannathan
Purdue University

PURDUE UNIVERSITY

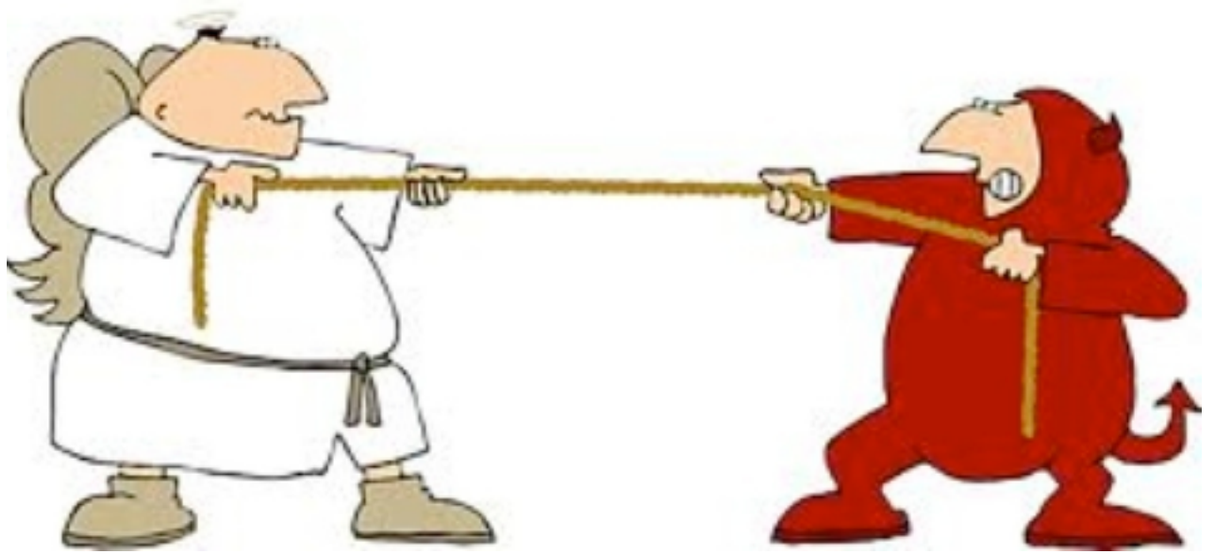
MultiMLton

Introduction

Two often competing goals when ***designing*** and ***implementing*** concurrency abstractions

Introduction

Two often competing goals when ***designing*** and ***implementing*** concurrency abstractions

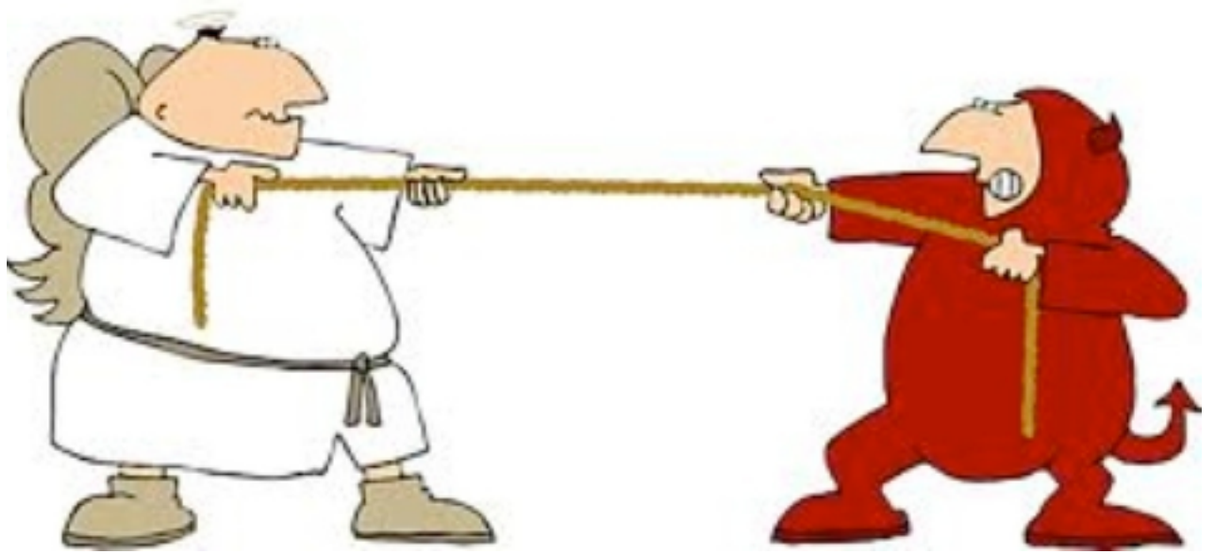


Simplicity
Safety

Performance
Functionality

Introduction

Two often competing goals when ***designing*** and ***implementing*** concurrency abstractions



Simplicity
Safety

Performance
Functionality

Always desirable to marry the two whenever possible

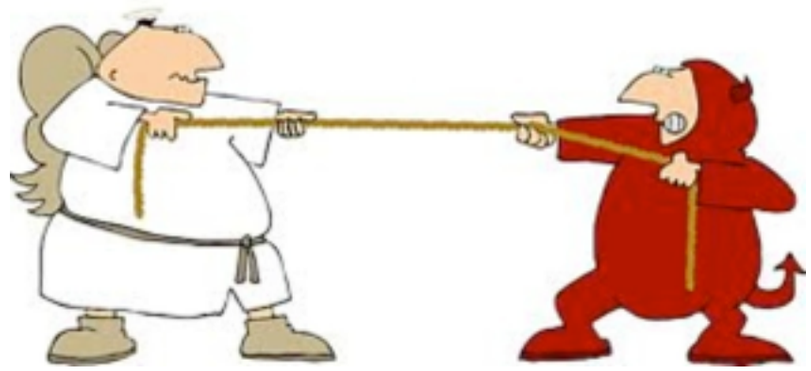
Big Picture

Big Picture

- Functional language + Synchronous message passing
  - ★ Communication = Data transfer + Synchronization

Big Picture

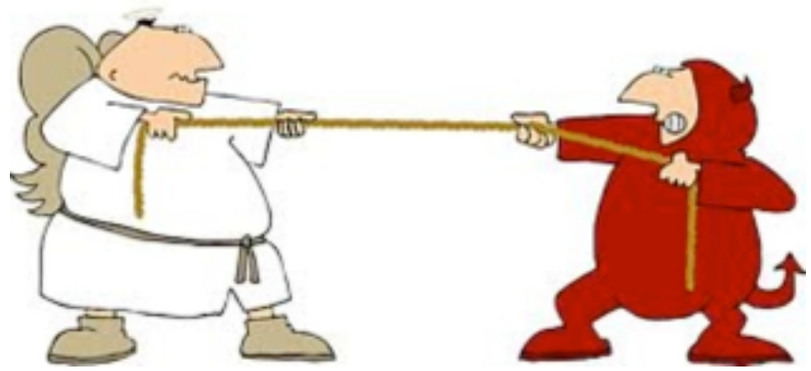
- Functional language + Synchronous message passing
  - ★ Communication = Data transfer + Synchronization
- However, in the cloud,



Synchrony *latency*

Big Picture

- Functional language + Synchronous message passing
  - ★ Communication = Data transfer + Synchronization
- However, in the cloud,

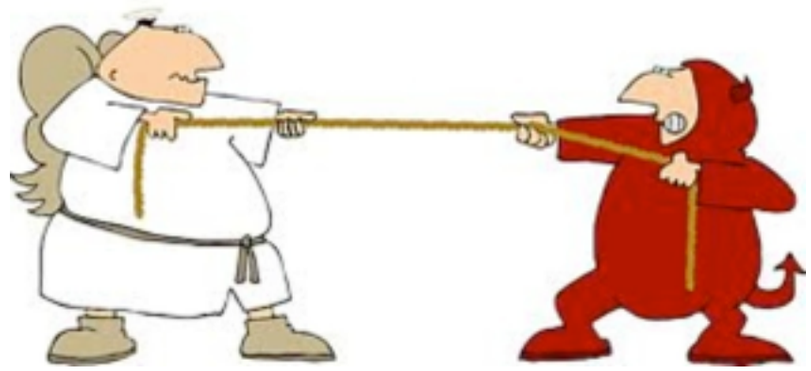


Synchrony *latency*

★ Explicit asynchrony complicates reasoning

Big Picture

- Functional language + Synchronous message passing
  - ★ Communication = Data transfer + Synchronization
- However, in the cloud,



Synchrony *latency*

  - ★ Explicit asynchrony complicates reasoning

Can we discharge synchronous communications ***asynchronously*** *while ensuring* ***observable equivalence****?*

Goal

Goal

1. Formalize the conditions under which the following equivalence holds:

$$[\![\, \texttt{send}\,(c, v)]\!]k \;\equiv\; [\![\, \texttt{asend}\,(c, v)]\!]k$$

Goal

1. Formalize the conditions under which the following equivalence holds:

$$[\![\,\texttt{send}\,(c, v)]\!]k \;\equiv\; [\![\,\texttt{asend}\,(c, v)]\!]k$$

2. A cloud infrastructure + speculative execution framework

   a. discharges synchronous sends asynchronously

   b. detects when the equivalence fails, and

   c. repairs failed executions

Context

Context

- A distributed extension of MultiMLton - MLton for scalable architectures

Context

- A distributed extension of MultiMLton - MLton for scalable architectures
- Concurrent ML
  - ★ Dynamic lightweight threads
  - ★ *Synchronous* message passing
  - ★ First-class events
    - ✦ *Composable synchronous protocols*

Context

- A distributed extension of MultiMLton - MLton for scalable architectures
- Concurrent ML
  - ★ Dynamic lightweight threads
  - ★ *Synchronous* message passing
  - ★ First-class events
    - ✦ *Composable synchronous protocols*

```
val channel : unit -> ‘a chan
val spawn   : (unit -> unit) -> thread_id
val send    : ‘a chan * ‘a -> unit
val recv    : ‘a chan -> ‘a
val sendEvt : ‘a chan * ‘a -> unit event
val recvEvt : ‘a chan -> ‘a event
val sync    : ‘a event -> ‘a
```

```
val never     : ‘a event
val alwaysEvt : ‘a -> ‘a event
val wrap  : ‘a event -> (‘a -> ‘b) ->
            ‘b event
val guard : (unit -> ‘a event) -> ‘a event
val choose : ‘a event list -> ‘a event
...
```

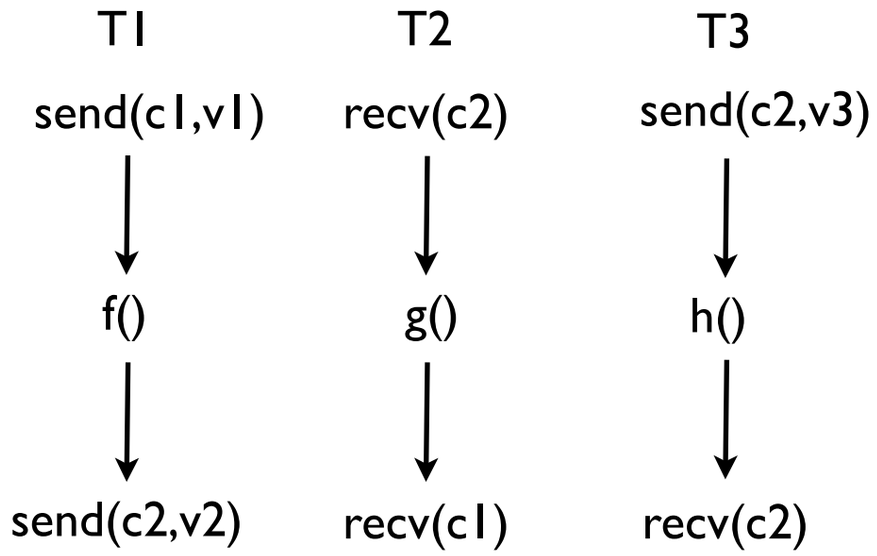
PURDUE UNIVERSITY

MultiMLton

Basic Idea (1)

T1	T2	T3
send(c1,v1) f() send(c2,v2)	recv(c2) g() recv(c1)	send(c2,v3) h() recv(c2)

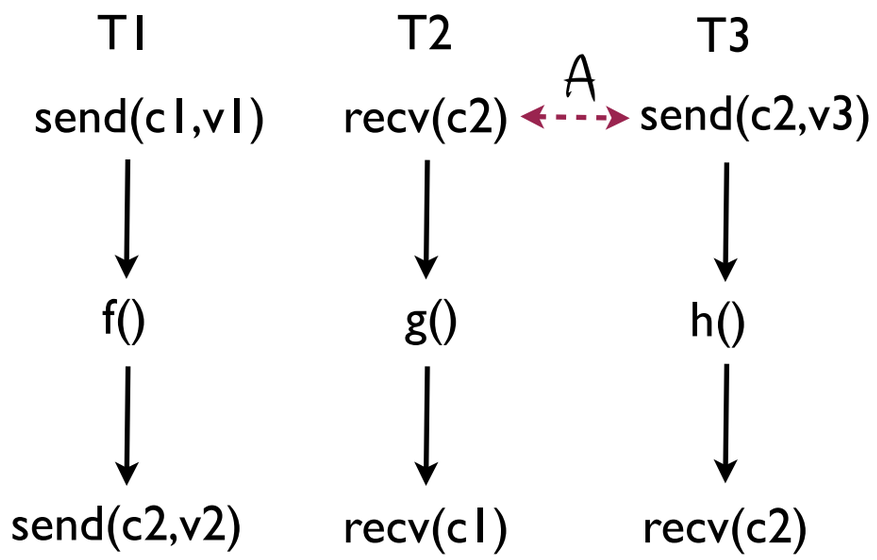
Synchronous Execution



Basic Idea (1)

```
        T1                  T2                  T3
send(c1,v1)          recv(c2)            send(c2,v3)
f()                  g()                 h()
send(c2,v2)          recv(c1)            recv(c2)
```

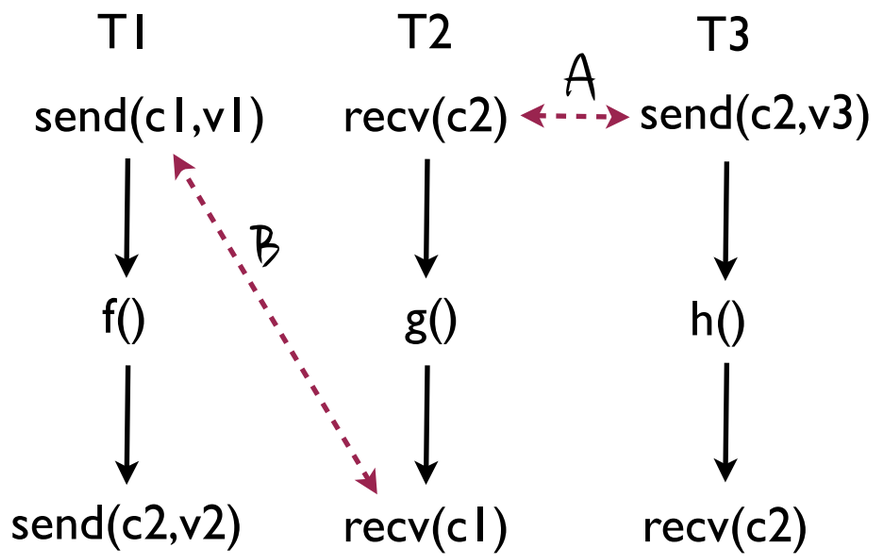
Synchronous Execution



Basic Idea (1)

```
        T1                    T2                    T3
send(c1,v1)          ||  recv(c2)          ||  send(c2,v3)
f()                  ||  g()               ||  h()
send(c2,v2)          ||  recv(c1)          ||  recv(c2)
```

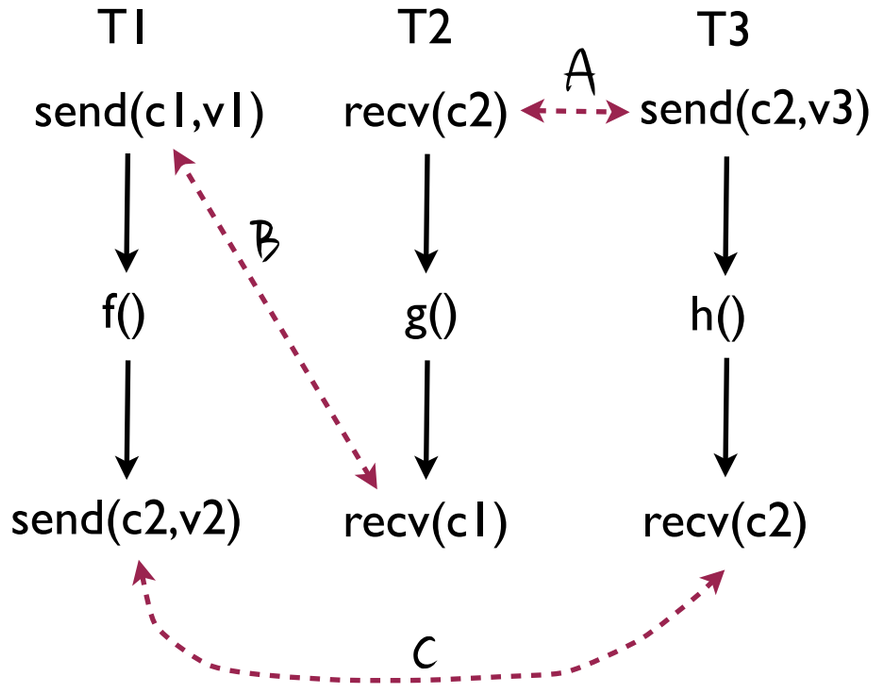
Synchronous Execution



Basic Idea (1)

T1	T2	T3
send(c1,v1) f() send(c2,v2)	recv(c2) g() recv(c1)	send(c2,v3) h() recv(c2)

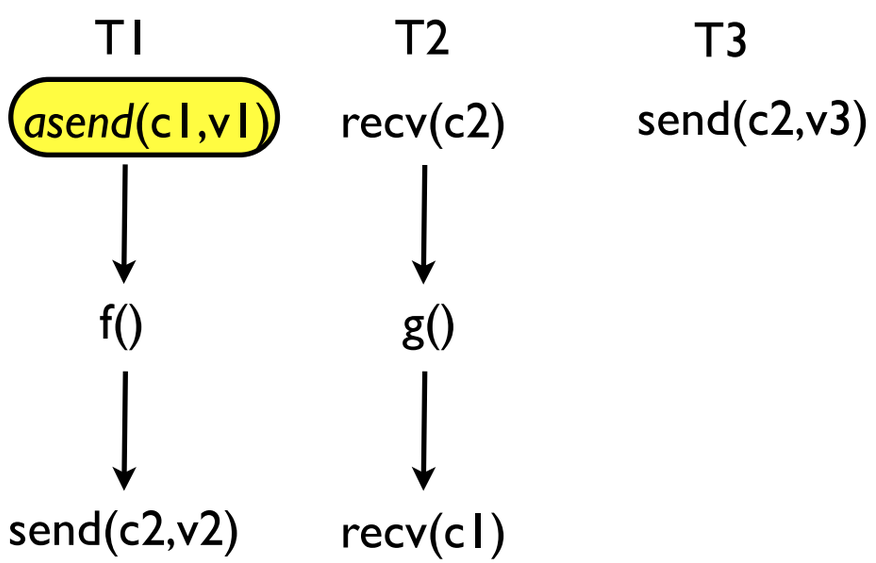
Synchronous Execution



Basic Idea (2)

T1	T2	T3
send(c1,v1) f() send(c2,v2)	recv(c2) g() recv(c1)	send(c2,v3) h() recv(c2)

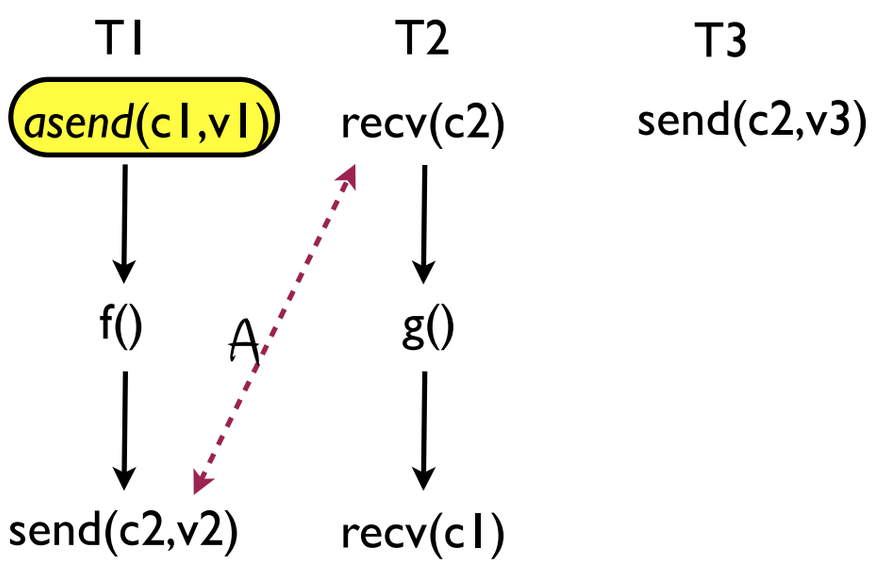
Asynchronous Execution



Basic Idea (2)

T1	T2	T3
send(c1,v1)	recv(c2)	send(c2,v3)
f()	g()	h()
send(c2,v2)	recv(c1)	recv(c2)

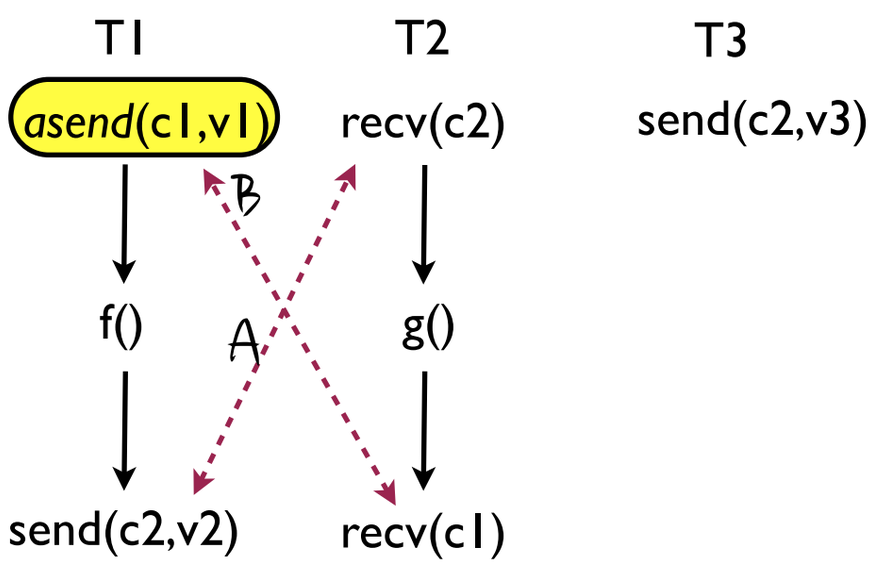
Asynchronous Execution



Basic Idea (2)

T1	T2	T3
send(c1,v1)	recv(c2)	send(c2,v3)
f()	g()	h()
send(c2,v2)	recv(c1)	recv(c2)

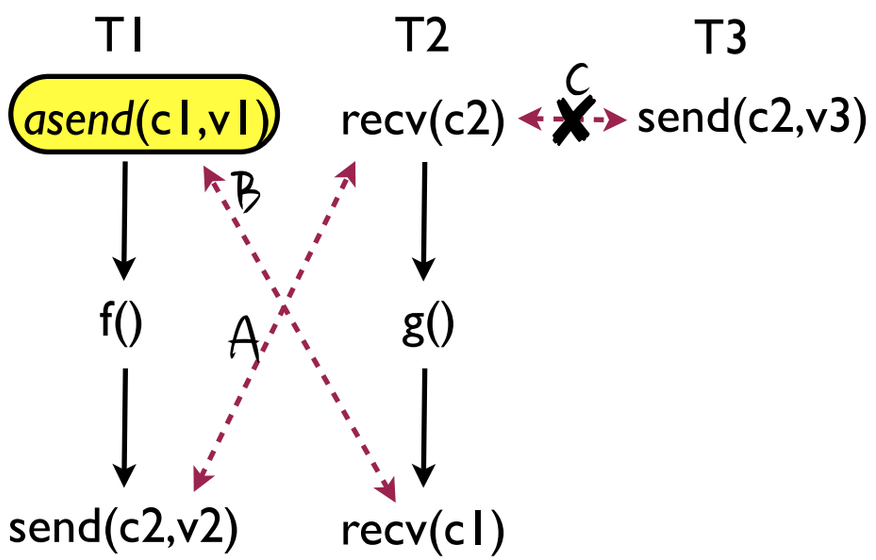
Asynchronous Execution



Basic Idea (2)

T1	T2	T3
send(c1,v1) f() send(c2,v2)	recv(c2) g() recv(c1)	send(c2,v3) h() recv(c2)

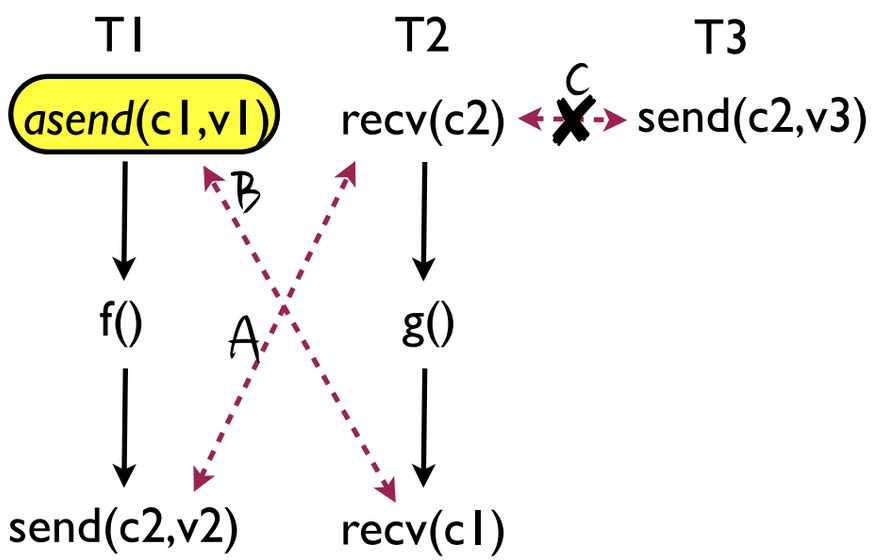
Asynchronous Execution



Basic Idea (2)

T1	T2	T3
send(c1,v1) f() send(c2,v2)	recv(c2) g() recv(c1)	send(c2,v3) h() recv(c2)

Asynchronous Execution



- Synchronous evaluation does not result in cyclic dependence
  - ★ Cyclic dependence => divergent behavior w.r.t synchronous evaluation

Example: Distributed Group Chat

Example: Distributed Group Chat

- No central server + Preserve causal dependence

Example: Distributed Group Chat

- No central server + Preserve causal dependence
  - ★ **Causal broadcast primitive**: If message A is generated as a response to message B, then A is delivered after B at every site

Example: Distributed Group Chat

- No central server + Preserve causal dependence
  - ★ **Causal broadcast primitive**: If message A is generated as a response to message B, then A is delivered after B at every site
  - ★ Asynchronous messaging => explicitly manage vector clocks and buffering on receiver side

Example: Distributed Group Chat

- No central server + Preserve causal dependence
  - ★ **Causal broadcast primitive**: If message A is generated as a response to message B, then A is delivered after B at every site
  - ★ Asynchronous messaging => explicitly manage vector clocks and buffering on receiver side
  - ★ Synchronous messaging => directly using point-to-point messaging

Example: Distributed Group Chat

- No central server + Preserve causal dependence
  - ★ **Causal broadcast primitive**: If message A is generated as a response to message B, then A is delivered after B at every site
  - ★ Asynchronous messaging => explicitly manage vector clocks and buffering on receiver side
  - ★ Synchronous messaging => directly using point-to-point messaging

```
fun bsend (BCHAN (vcList, acList), v: 'a, id: int) : unit =
let
  val _ = map (fn vc => if (vc = nth (vcList, id)) then () else send (vc, v))
              vcList (* phase 1 -- Value distribution *)
  val _ = map (fn ac => if (ac = nth (acList, id)) then () else recv ac)
              acList (* phase 2 -- Acknowledgments *)
in ()
end
```

synchronously send values

prevent receivers from proceeding until all members have received the value

Example: Distributed Group Chat

- No central server + Preserve causal dependence
  - ★ **Causal broadcast primitive**: If message A is generated as a response to message B, then A is delivered after B at every site
  - ★ Asynchronous messaging => explicitly manage vector clocks and buffering on receiver side
  - ★ Synchronous messaging => directly using point-to-point messaging

```
fun bsend (BCHAN (vcList, acList), v: 'a, id: int) : unit =
let
  val _ = map (fn vc => if (vc = nth (vcList, id)) then () else send (vc, v))
              vcList (* phase 1 -- Value distribution *)
  val _ = map (fn ac => if (ac = nth (acList, id)) then () else recv ac)
              acList (* phase 2 -- Acknowledgments *)
in ()
end
```

synchronously send values

prevent receivers from proceeding until all members have received the value

- Simple but likely to be inefficient - phase 2 is a global barrier!

Example: Distributed Group Chat

- No central server + Preserve causal dependence
  - ★ **Causal broadcast primitive**: If message A is generated as a response to message B, then A is delivered after B at every site
  - ★ Asynchronous messaging => explicitly manage vector clocks and buffering on receiver side
  - ★ Synchronous messaging => directly using point-to-point messaging

```
fun bsend (BCHAN (vcList, acList), v: 'a, id: int) : unit =
let
  val _ = map (fn vc => if (vc = nth (vcList, id)) then () else send (vc, v))
              vcList (* phase 1 -- Value distribution *)
  val _ = map (fn ac => if (ac = nth (acList, id)) then () else recv ac)
              acList (* phase 2 -- Acknowledgments *)
in ()
end
```

synchronously send values

prevent receivers from proceeding until all members have received the value

- Simple but likely to be inefficient - phase 2 is a global barrier!
  - ★ Discharging asynchronously breaks causal ordering

Example: Distributed Group Chat

- No central server + Preserve causal dependence
  - ★ **Causal broadcast primitive**: If message A is generated as a response to message B, then A is delivered after B at every site
  - ★ Asynchronous messaging => explicitly manage vector clocks and buffering on receiver side
  - ★ Synchronous messaging => directly using point-to-point messaging

```
fun bsend (BCHAN (vcList, acList), v: 'a, id: int) : unit =
let
  val _ = map (fn vc => if (vc = nth (vcList, id)) then () else send (vc, v))
              vcList (* phase 1 -- Value distribution *)
  val _ = map (fn ac => if (ac = nth (acList, id)) then () else recv ac)
              acList (* phase 2 -- Acknowledgments *)
in ()
end
```

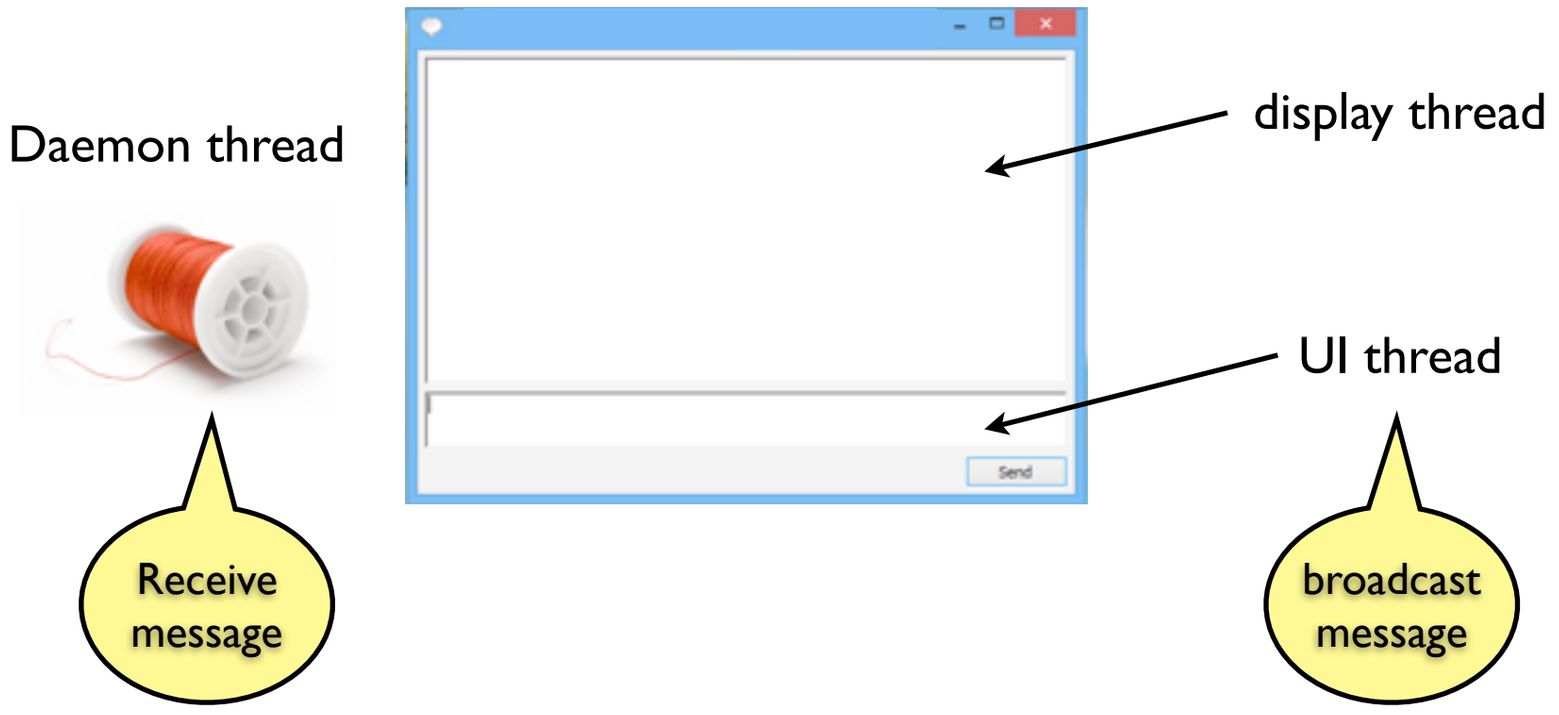
synchronously send values

prevent receivers from proceeding until all members have received the value

- Simple but likely to be inefficient - phase 2 is a global barrier!
  - ★ Discharging asynchronously breaks causal ordering
  - ★ *Our idea: program synchronously, discharge asynchronously, detect and remediate causal ordering violations*

Example: Distributed Group Chat

- A distributed group chat program = {Node}
- Node = MultiMLton process = {CML threads}



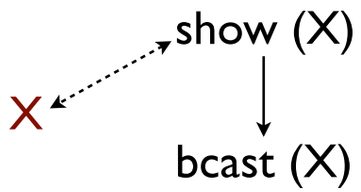
Distributed Group Chat - Run 1

Display	UI	Daemon

Display	UI	Daemon

Distributed Group Chat - Run 1

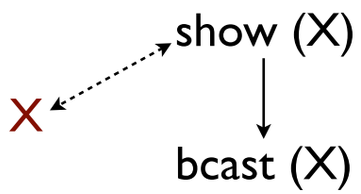
Display	UI	Daemon



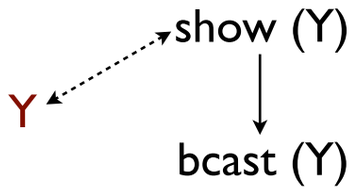
Display	UI	Daemon

Distributed Group Chat - Run 1

Display	UI	Daemon



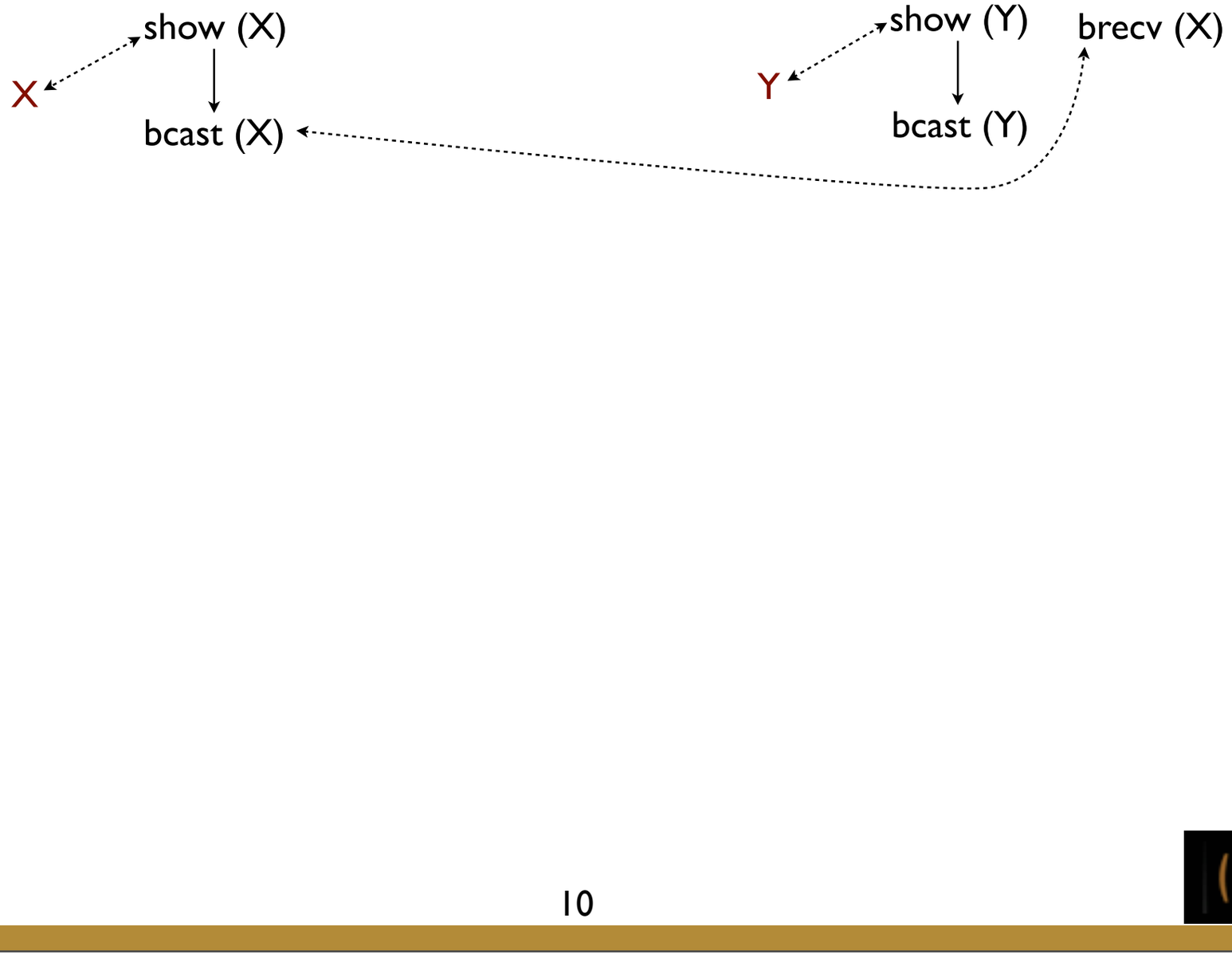
Display	UI	Daemon



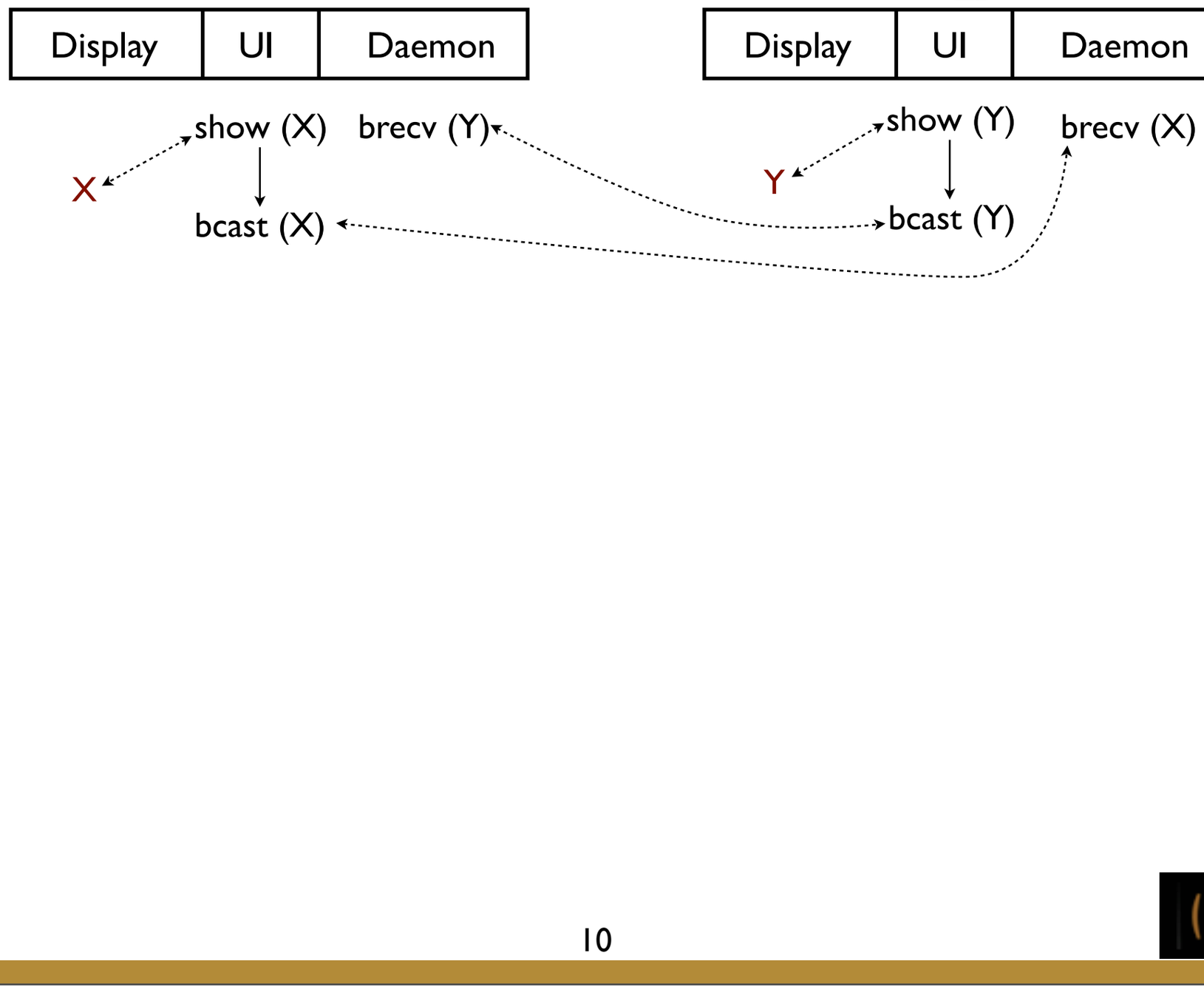
Distributed Group Chat - Run 1

Display	UI	Daemon

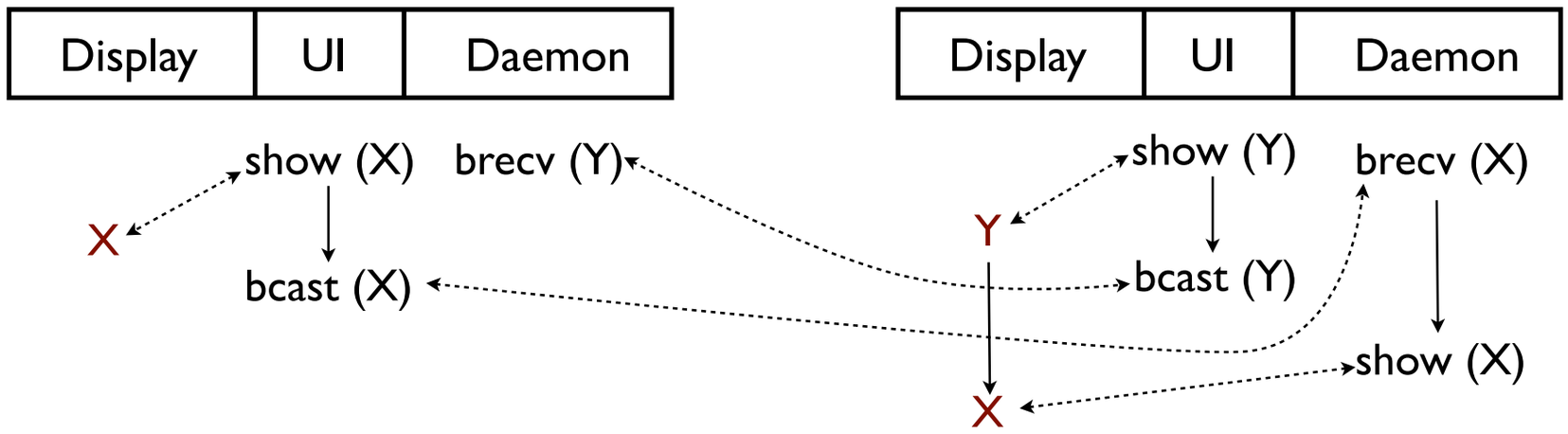
Display	UI	Daemon



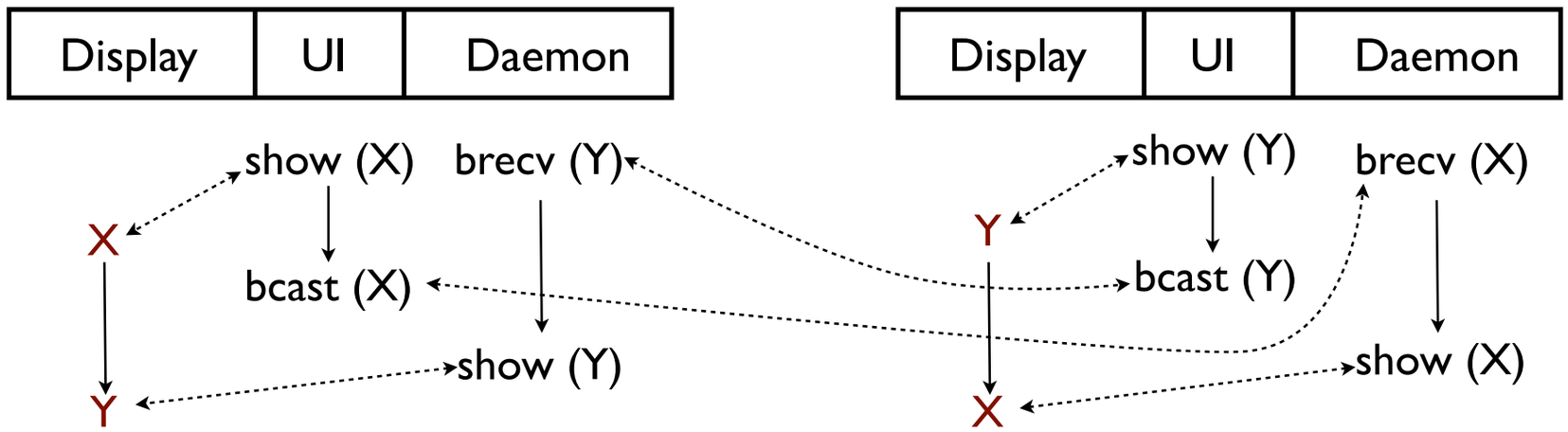
Distributed Group Chat - Run 1



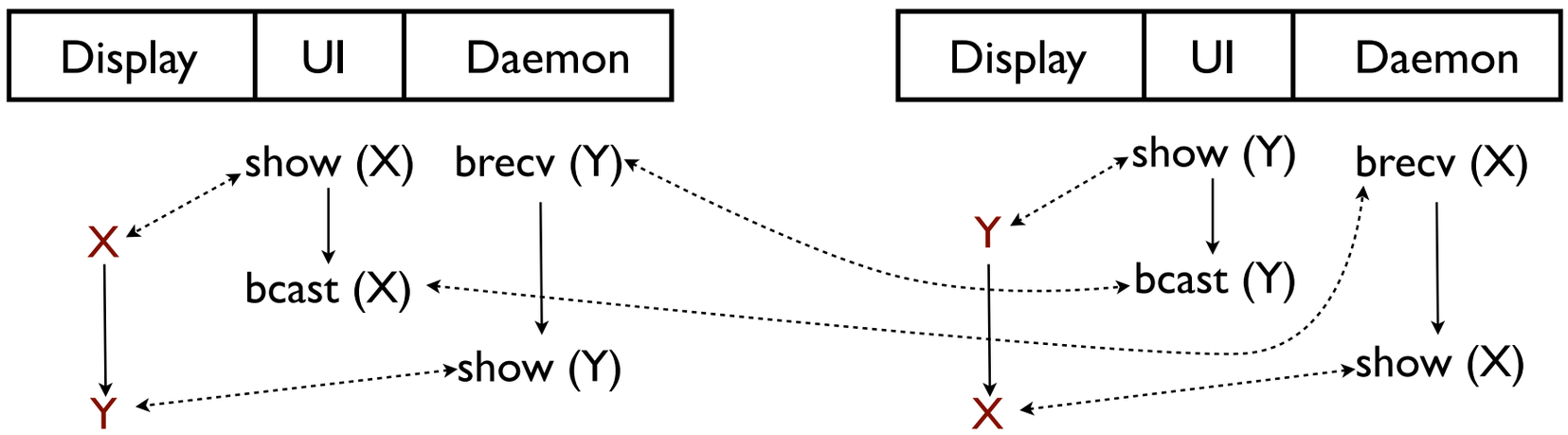
Distributed Group Chat - Run 1



Distributed Group Chat - Run 1



Distributed Group Chat - Run 1



- Observations
  - ★ X and Y independently generated => No causal dependence between bcast (X) and bcast (Y)
- No Cycles => Correct execution!

Distributed Group Chat - Run 2

Display	UI	Daemon

Display	UI	Daemon

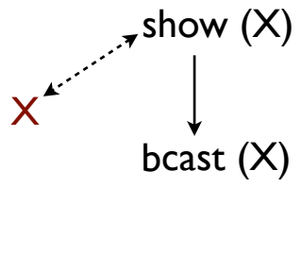
Display	UI	Daemon

Distributed Group Chat - Run 2

Display	UI	Daemon

Display	UI	Daemon

Display	UI	Daemon

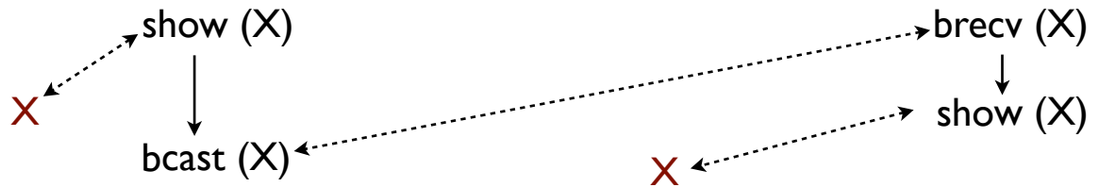


Distributed Group Chat - Run 2

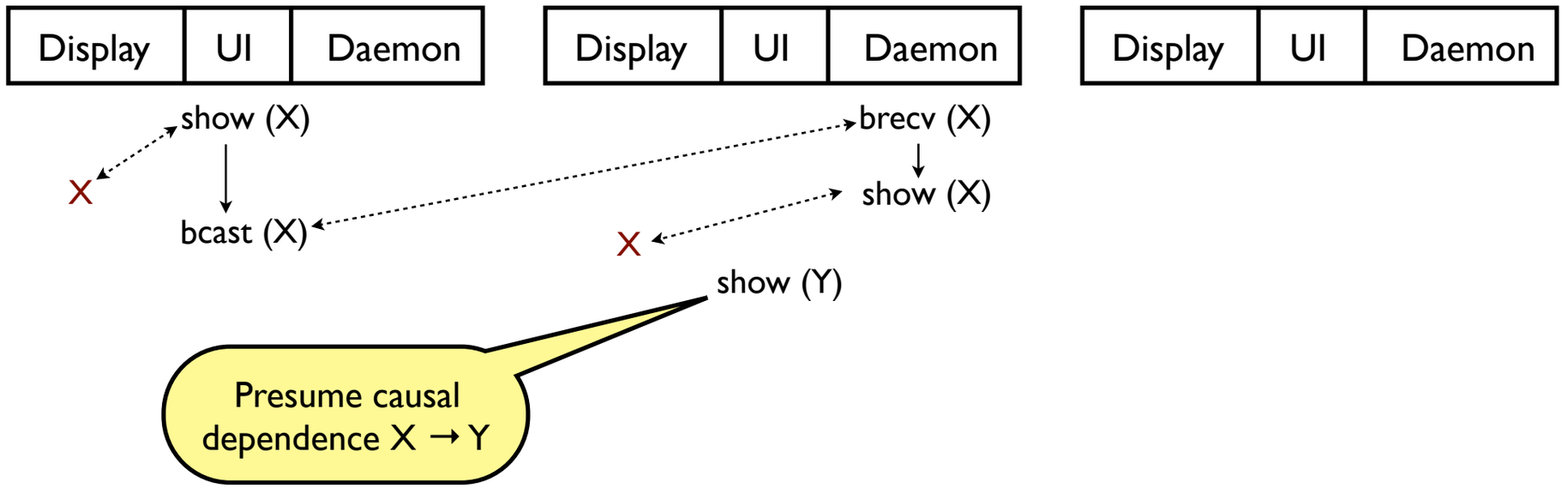
Display	UI	Daemon

Display	UI	Daemon

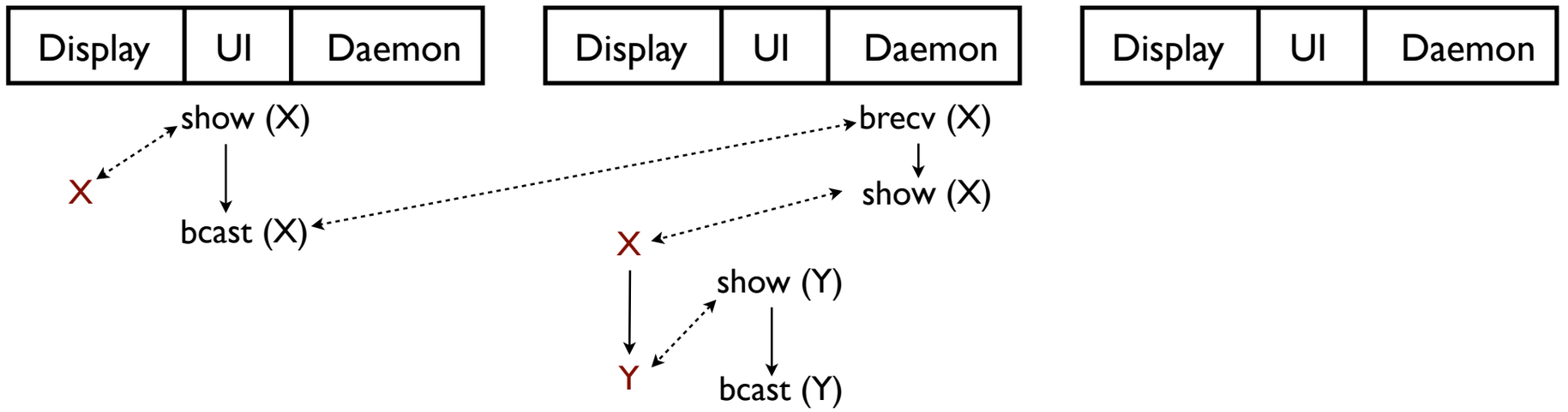
Display	UI	Daemon



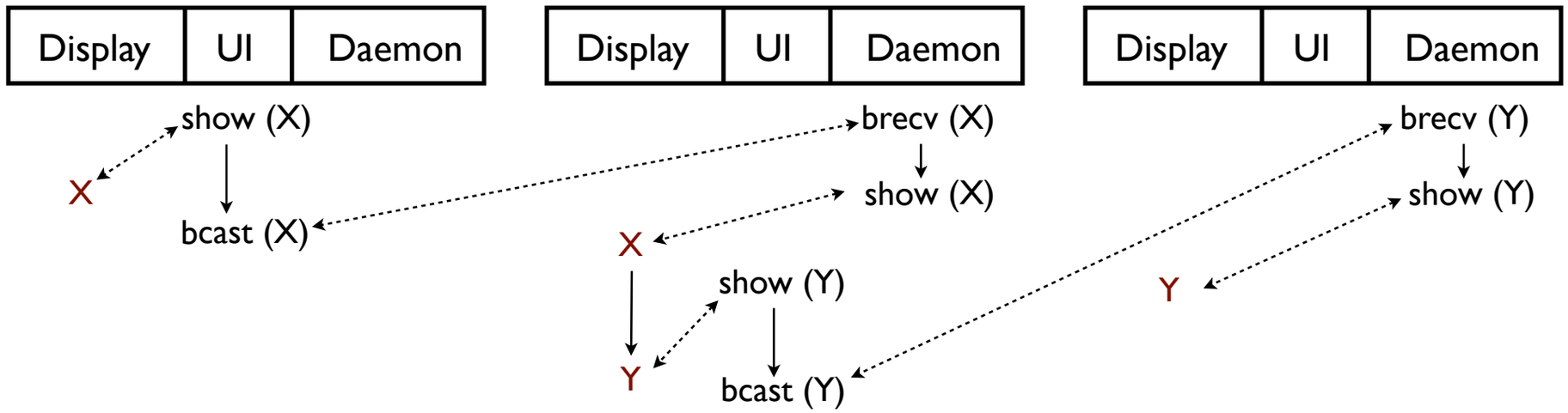
Distributed Group Chat - Run 2



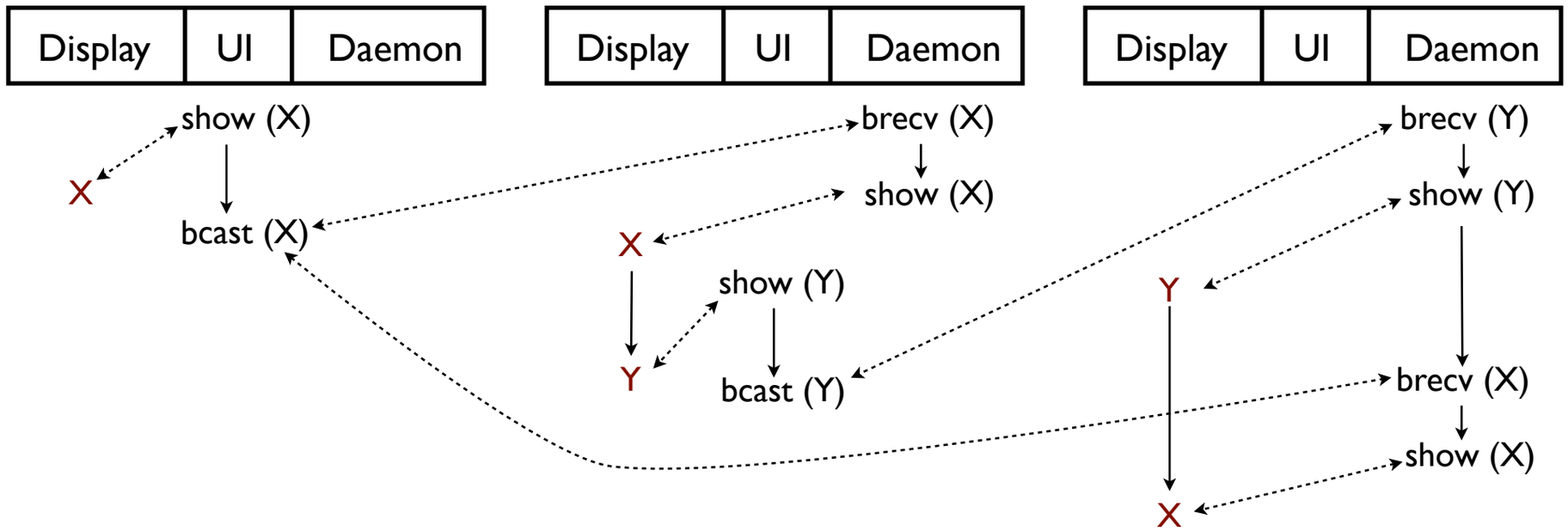
Distributed Group Chat - Run 2



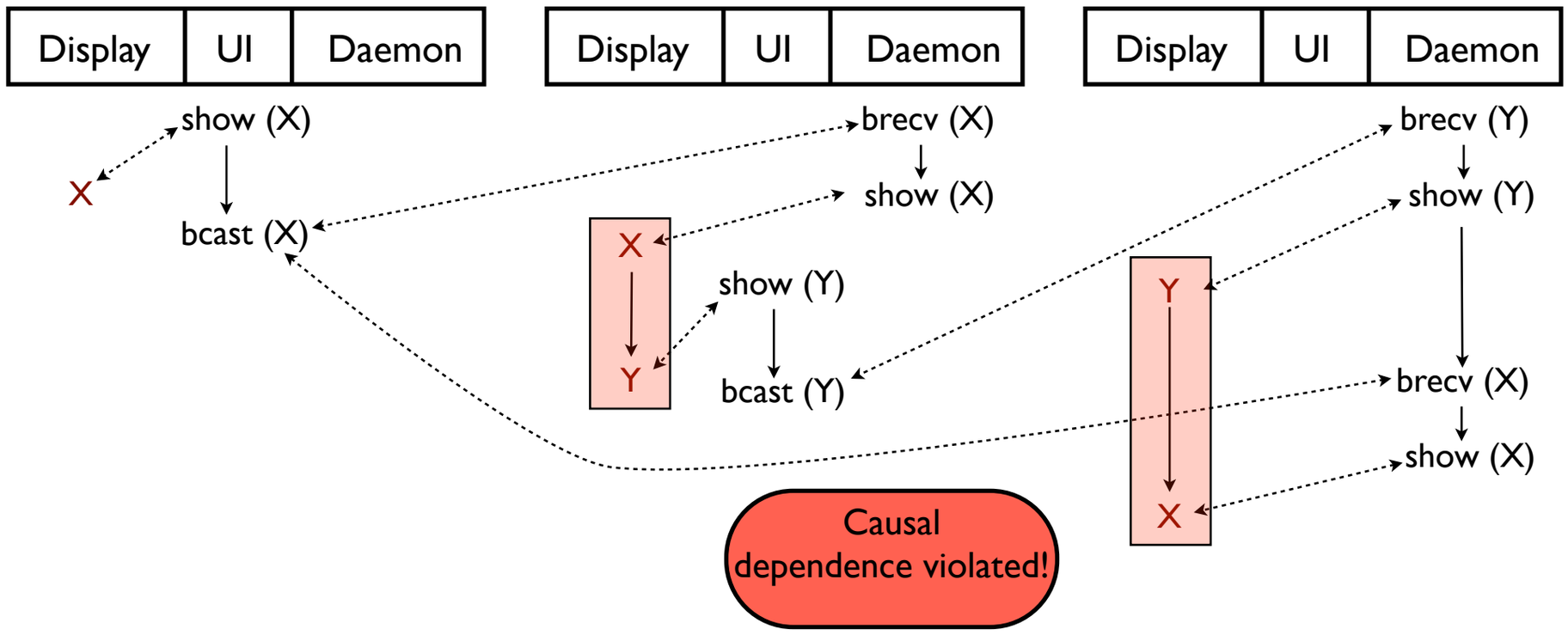
Distributed Group Chat - Run 2



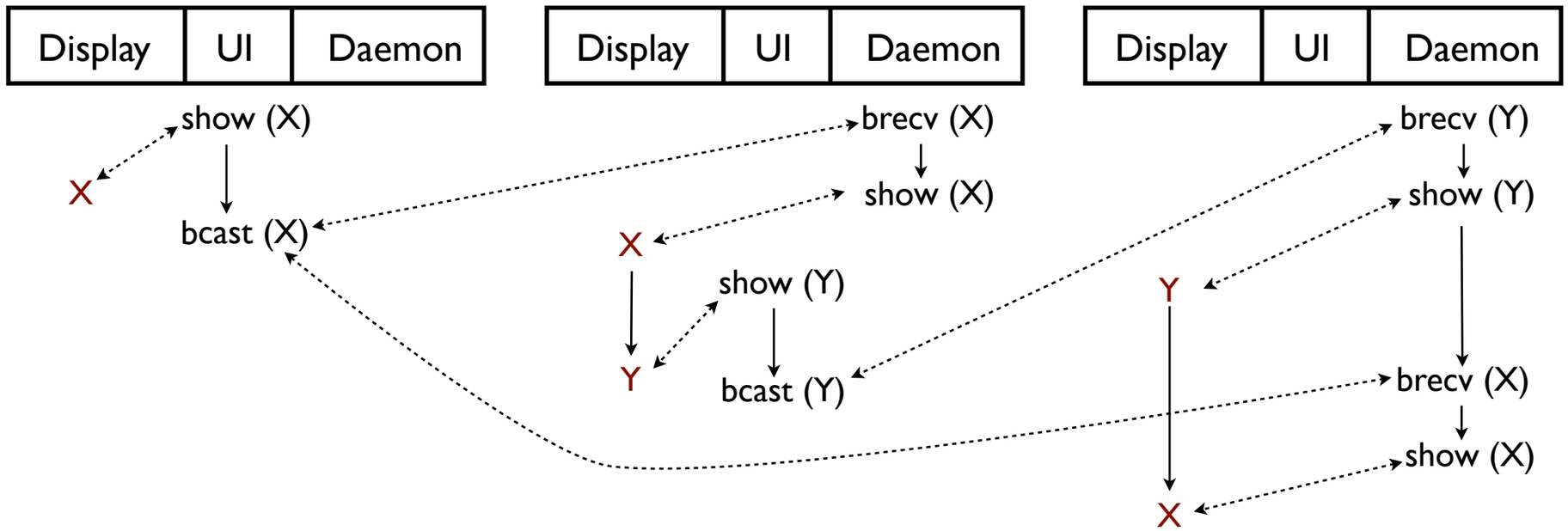
Distributed Group Chat - Run 2



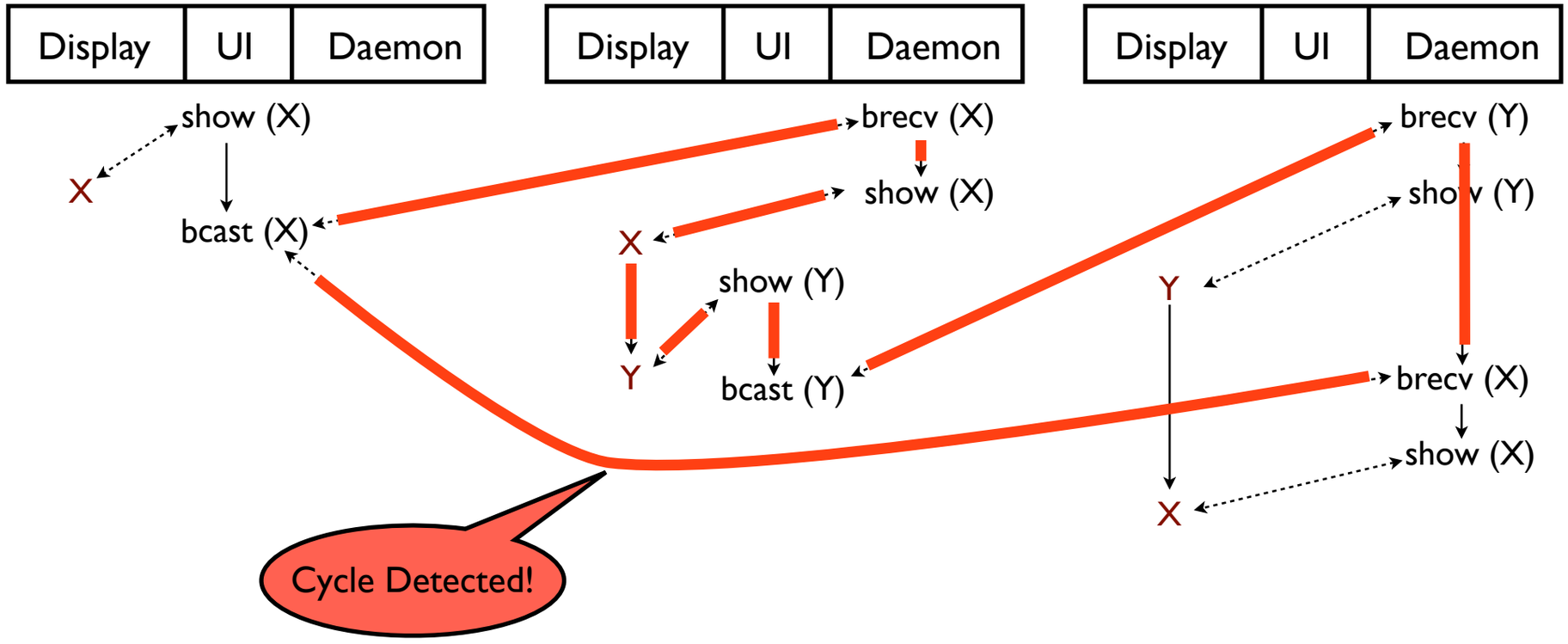
Distributed Group Chat - Run 2



Distributed Group Chat - Run 2



Distributed Group Chat - Run 2



Distributed Group Chat - Results

- Simulation on 3 geo-distributed Amazon EC2 instances
- Measure time between message initiation and receipt by all parties over 1000 iterations

Execution	*Avg.time (ms)*	*Errors*
Sync	1540	0
Unsafe Async	520	7
Safe Async ($\mathbb{R}x^{\mathrm{CML}}$)	533	0

Formalization Overview

Formalization Overview

- Reason ***axiomatically*** about executions (relaxed or otherwise)
  - ★ Similar to formalizations used in relaxed memory models
  - ★ Declarative characterization of (relaxed) CML behavior

Formalization Overview

- Reason ***axiomatically*** about executions (relaxed or otherwise)
  - ★ Similar to formalizations used in relaxed memory models
  - ★ Declarative characterization of (relaxed) CML behavior
- ***Actions + happens-before relation***
  - ★ Captures visibility and dependence properties

Formalization Overview

- Reason ***axiomatically*** about executions (relaxed or otherwise)
  - ★ Similar to formalizations used in relaxed memory models
  - ★ Declarative characterization of (relaxed) CML behavior
- ***Actions + happens-before relation***
  - ★ Captures visibility and dependence properties
- Happens-before is intentionally *relaxed*: may define more behaviors than possible in CML
  - ★ Strengthen the relation with ***well-formedness*** conditions

Actions and Execution

- Actions:

$$
\begin{array}{lcll}
\mathbb{A} & := & b_t & \text{(thread t starts)} \\
 & | & e_t & \text{(thread t ends)} \\
 & | & j_t^m t' & \text{(thread t detects thread t' has terminated)} \\
 & | & f_t^m t' & \text{(thread t creates a new thread t')} \\
 & | & s_t^m c, v & \text{(thread t sends value v on channel c} \\
 & | & r_t^m c & \text{(thread t receives a value on channel c)} \\
 & | & p_t^m v & \text{(thread t outputs an observable value v)}
\end{array}
$$

$$c \in \mathbb{C} \quad t, t' \in \mathbb{T} \quad v \in \mathbb{V} \quad m, n \in \mathbb{N}$$

- Execution:

$$\mathsf{E} := \langle \mathsf{P}, \mathsf{A}, \rightarrow_{po}, \mathsf{M} \rangle$$

program

a set of actions

program order: sequential actions of a thread

communication match: relates sends and receives

Communication and Thread Dependence

- A communication order exists between matching communication actions:

$$\alpha \rightarrow_{co} \beta \text{ and } \beta \rightarrow_{co} \alpha \text{ if } M(\alpha) = \beta$$

- Thread dependence order:

$$\alpha \rightarrow_{td} \beta \text{ if:}$$
$$(1)\ \alpha = f_t^m t' \text{ and } \beta = b_{t'} \text{ or}$$
$$(2)\ \alpha = e_t \text{ and } \beta = j_{t'}^m t$$

Happens-before Relation

- Establishes both intra- and inter-thread dependences:

$$
\begin{aligned}
\rightarrow_{hb} \quad = \quad & (\rightarrow_{po} \cup \rightarrow_{td} \cup \\
& \{(\alpha, \beta) \mid \alpha \rightarrow_{co} \alpha' \wedge \alpha' \rightarrow_{po} \beta\} \cup \\
& \{(\beta, \alpha) \mid \beta \rightarrow_{po} \alpha' \wedge \alpha' \rightarrow_{co} \alpha\})^{+}
\end{aligned}
$$

Happens-before Relation

- Establishes both intra- and inter-thread dependences:

$$\begin{aligned} \rightarrow_{hb} \quad = \quad & (\rightarrow_{po} \cup \rightarrow_{td} \cup \\ & \{(\alpha, \beta) \mid \alpha \rightarrow_{co} \alpha' \wedge \alpha' \rightarrow_{po} \beta\} \cup \\ & \{(\beta, \alpha) \mid \beta \rightarrow_{po} \alpha' \wedge \alpha' \rightarrow_{co} \alpha\})^{+} \end{aligned}$$

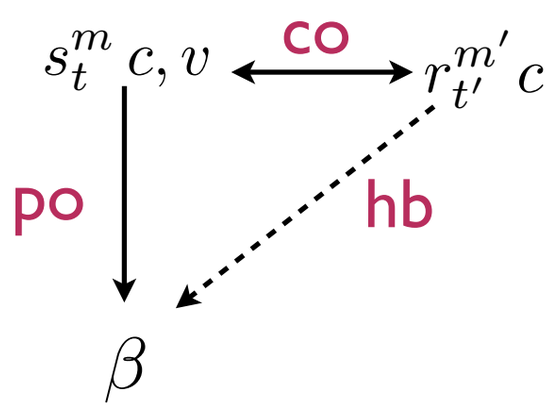
- Two actions not related by happens-before relation are said to be concurrent
  - ★ A send action and its matching receive action are concurrent!

Happens-before Relation

- Establishes both intra- and inter-thread dependences:

$$
\begin{aligned}
\rightarrow_{hb} \quad = \quad & (\rightarrow_{po} \cup \rightarrow_{td} \cup \\
& \{(\alpha, \beta) \mid \alpha \rightarrow_{co} \alpha' \wedge \alpha' \rightarrow_{po} \beta\} \cup \\
& \{(\beta, \alpha) \mid \beta \rightarrow_{po} \alpha' \wedge \alpha' \rightarrow_{co} \alpha\})^{+}
\end{aligned}
$$

- Two actions not related by happens-before relation are said to be concurrent
  - ★ A send action and its matching receive action are concurrent!



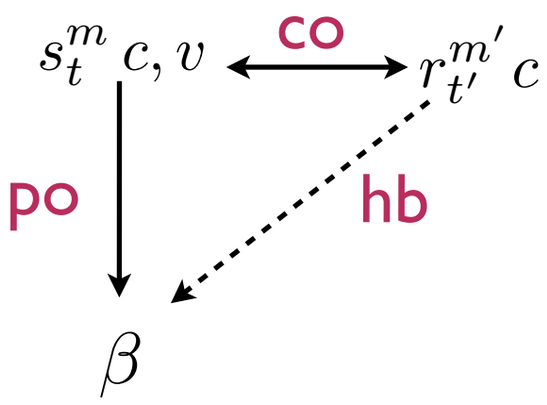
Happens-before Relation

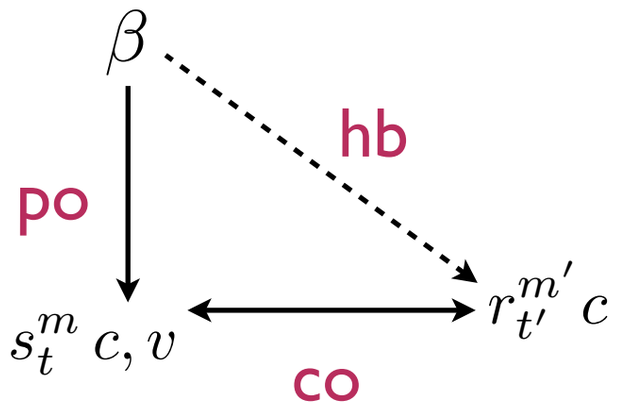
- Establishes both intra- and inter-thread dependences:

$$
\begin{aligned}
\rightarrow_{hb} \quad = \quad & (\rightarrow_{po} \cup \rightarrow_{td} \cup \\
& \{(\alpha, \beta) \mid \alpha \rightarrow_{co} \alpha' \wedge \alpha' \rightarrow_{po} \beta\} \cup \\
& \{(\beta, \alpha) \mid \beta \rightarrow_{po} \alpha' \wedge \alpha' \rightarrow_{co} \alpha\})^{+}
\end{aligned}
$$

- Two actions not related by happens-before relation are said to be concurrent

  ★ A send action and its matching receive action are concurrent!





Example

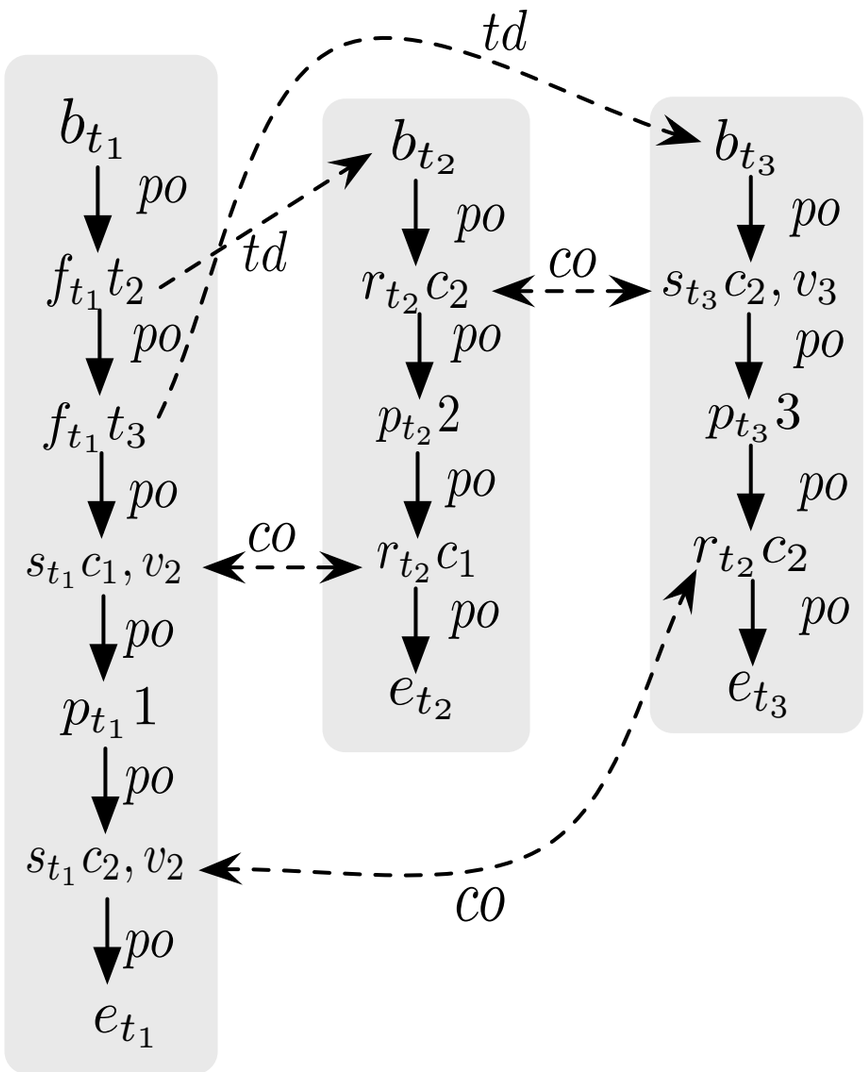
T1	T2	T3

`send(c1,v1)`
`f()`
`send(c2,v2)`
	`recv(c2)`
	`g()`
	`recv(c1)`
		`send(c2,v3)`
		`h()`
		`recv(c2)`

- Assume T1 spawns T2 and T3
- Let f, g, h = print 1, print 2, print 3

Example

T1	T2	T3
send(c1,v1) f() send(c2,v2)	recv(c2) g() recv(c1)	send(c2,v3) h() recv(c2)

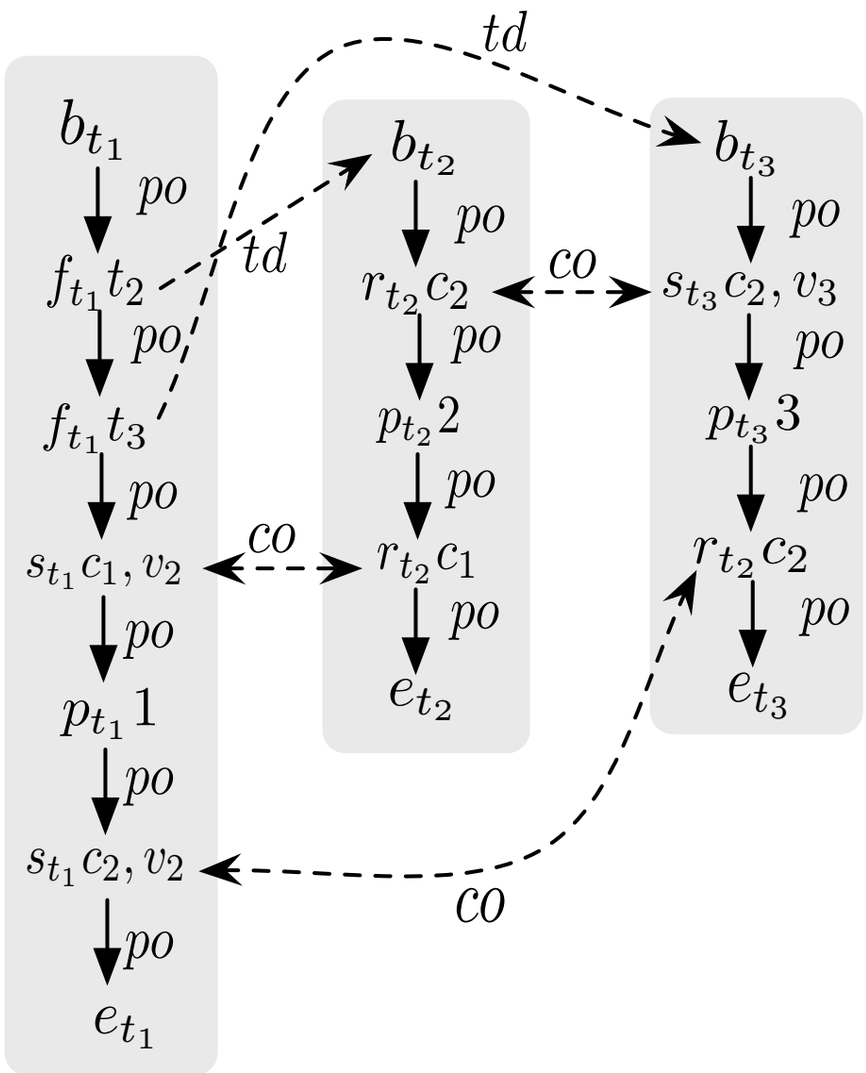


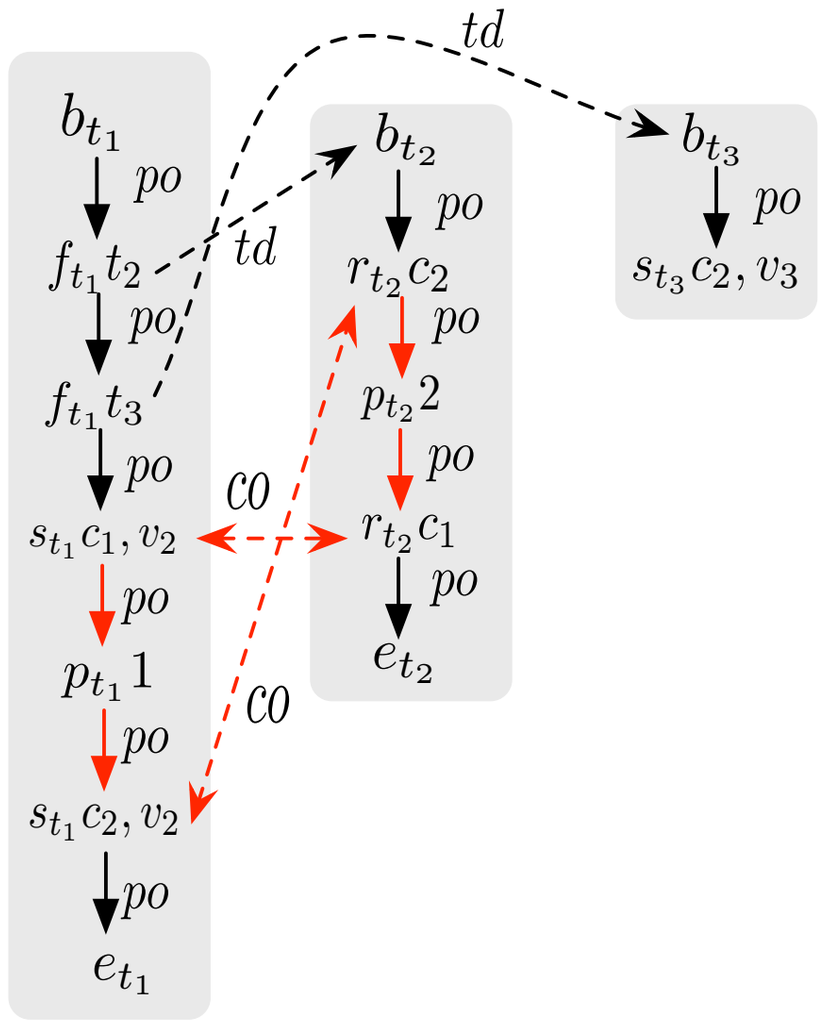
- Assume T1 spawns T2 and T3
- Let f, g, h = print 1, print 2, print 3

Example

```
T1              T2              T3
send(c1,v1)  ||  recv(c2)     ||  send(c2,v3)
f()          ||  g()          ||  h()
send(c2,v2)  ||  recv(c1)     ||  recv(c2)
```

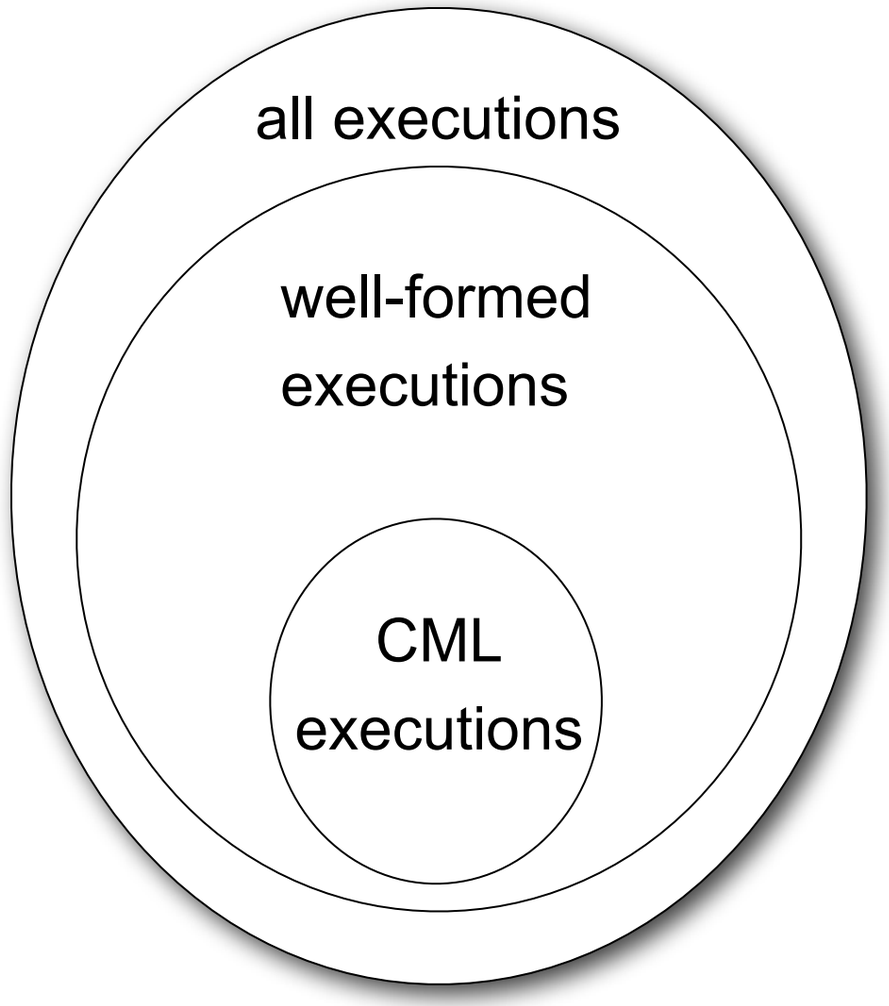
- Assume T1 spawns T2 and T3
- Let f, g, h = print 1, print 2, print 3





Well-formed Executions

Obs (Well-formed Execution of P) ∈ {Obs (CML Execution of P)}



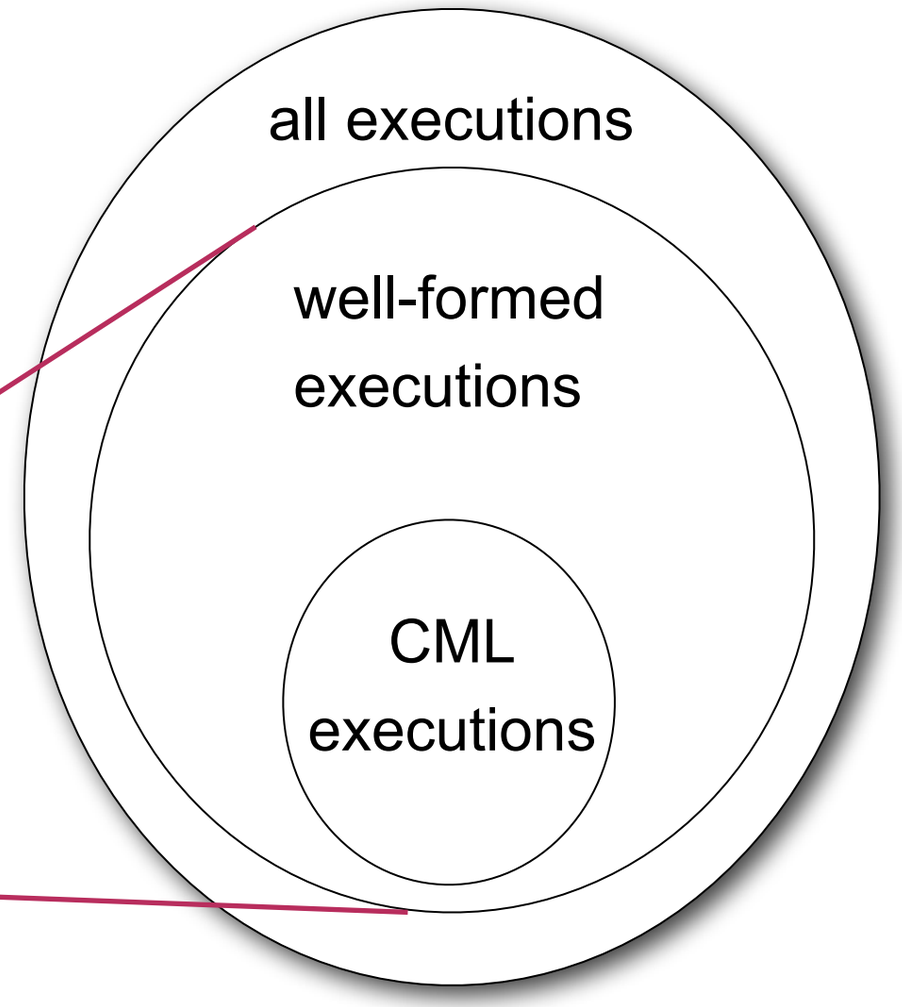
PURDUE UNIVERSITY

MultiMLton

Well-formed Executions

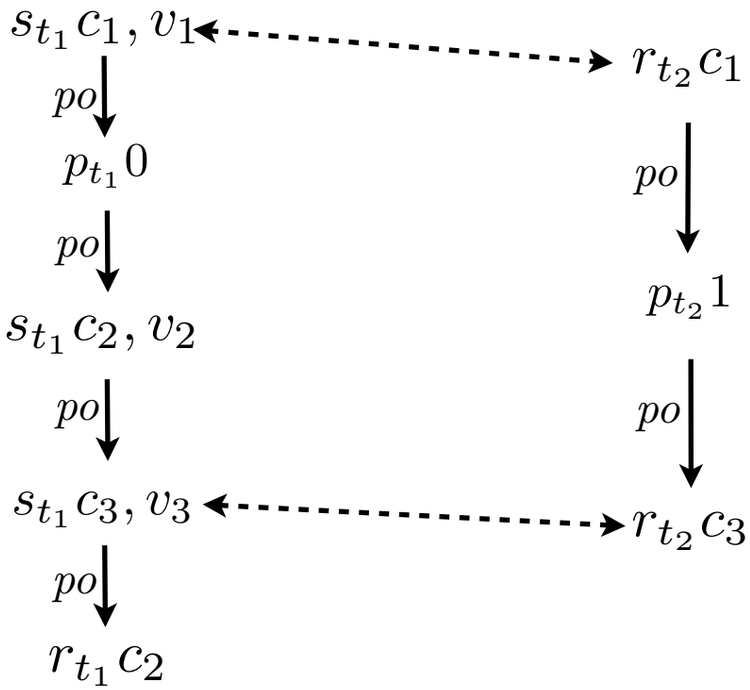
Obs (Well-formed Execution of P) ∈ {Obs (CML Execution of P)}

1. Sensible intra-thread semantics
2. Acyclic happens-before relation
3. No outstanding communication action preceding an observable action



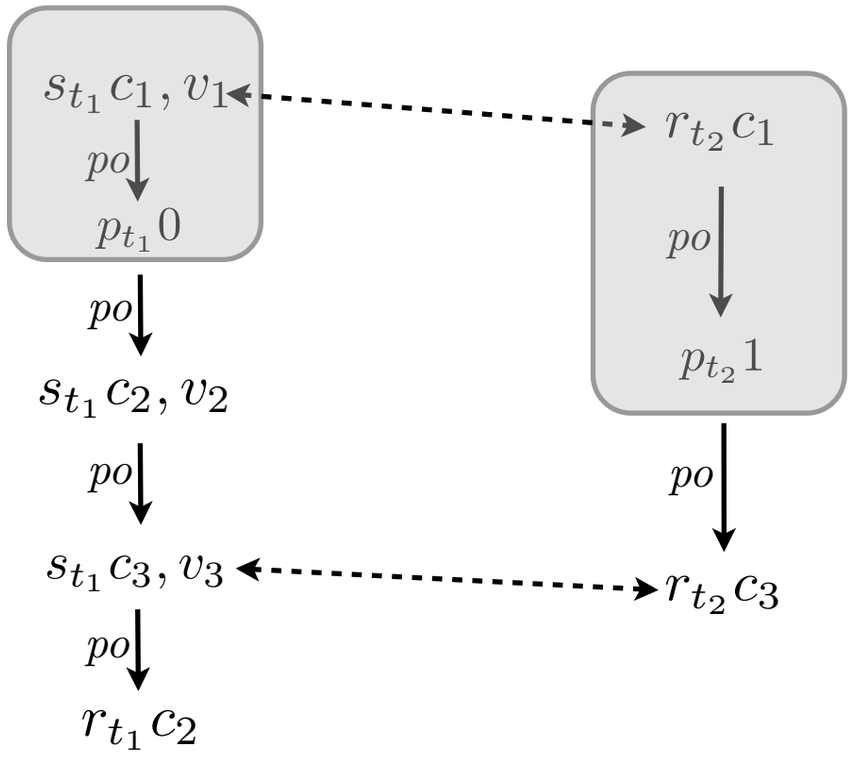
Well-formedness -> Speculation

Track executions to see if they become
ill-formed (rollback) or turn into CML executions (commit)



Well-formedness -> Speculation

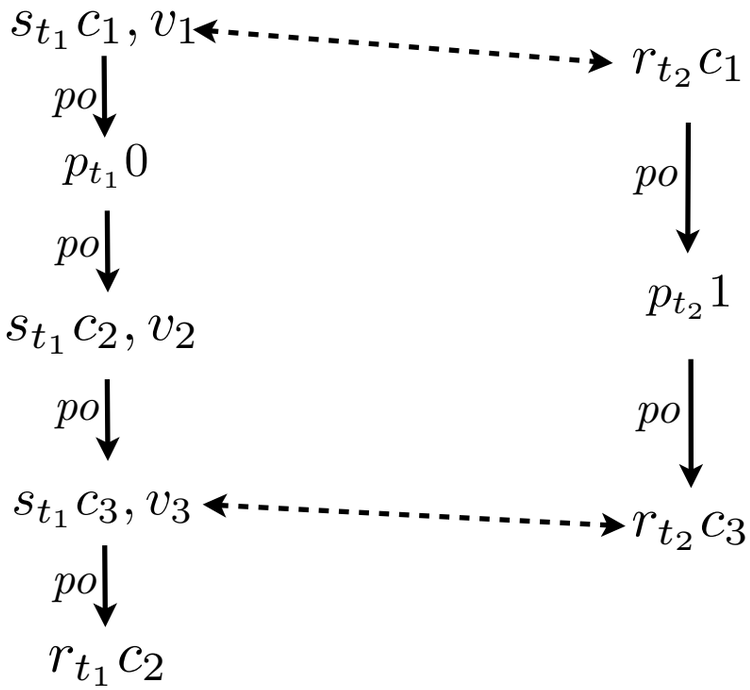
Track executions to see if they become
ill-formed (rollback) or turn into CML executions (commit)



A CML execution

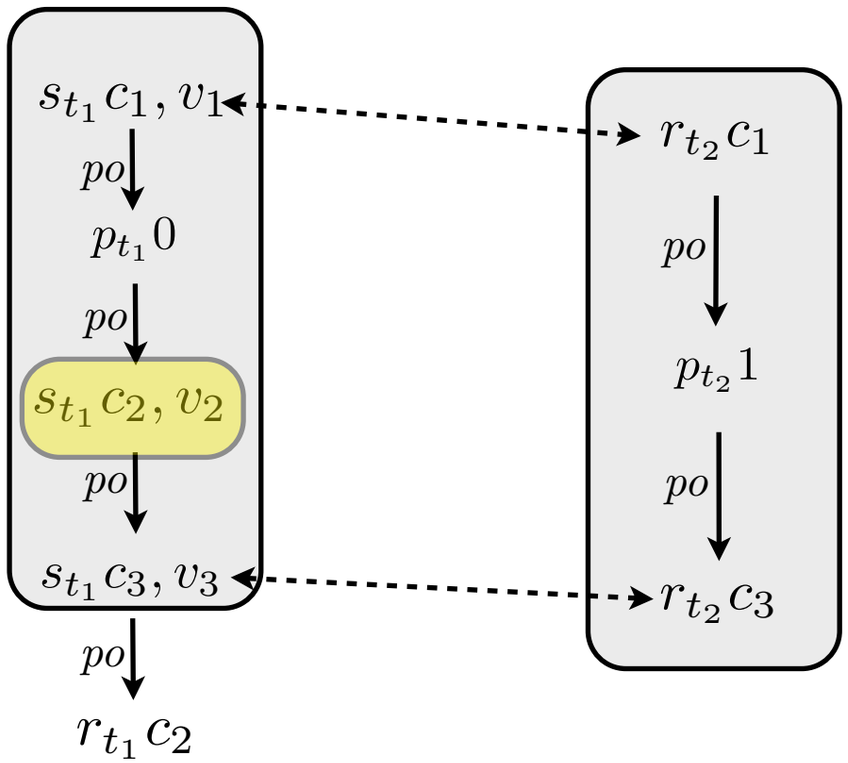
Well-formedness -> Speculation

Track executions to see if they become
ill-formed (rollback) or turn into CML executions (commit)



Well-formedness -> Speculation

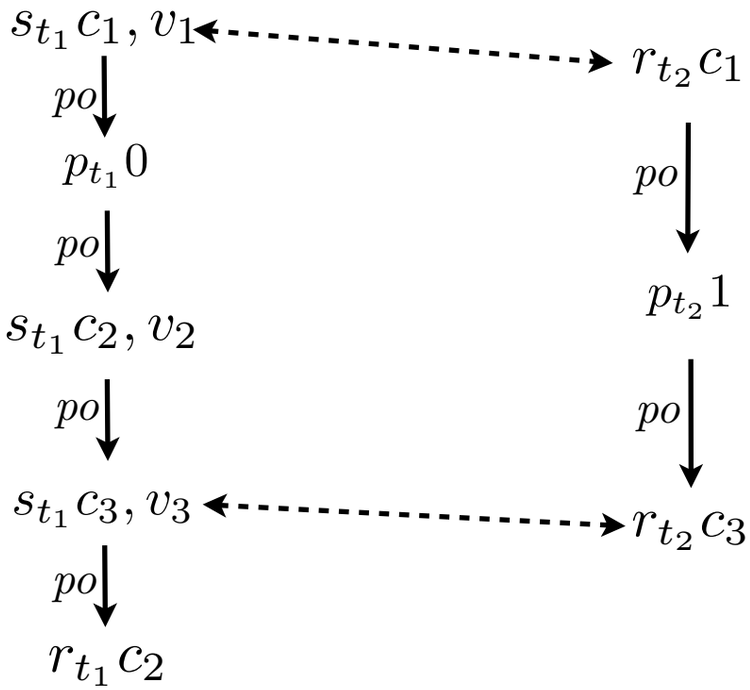
Track executions to see if they become
ill-formed (rollback) or turn into CML executions (commit)



A well-formed execution that *can lead* to a CML execution

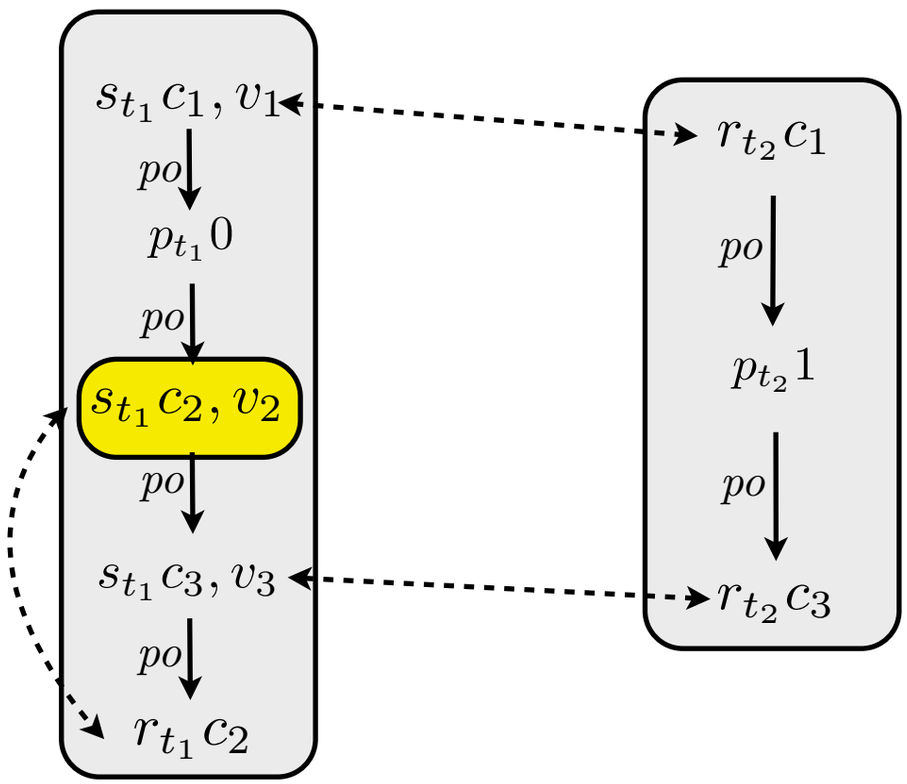
Well-formedness -> Speculation

Track executions to see if they become
ill-formed (rollback) or turn into CML executions (commit)



Well-formedness -> Speculation

Track executions to see if they become
ill-formed (rollback) or turn into CML executions (commit)

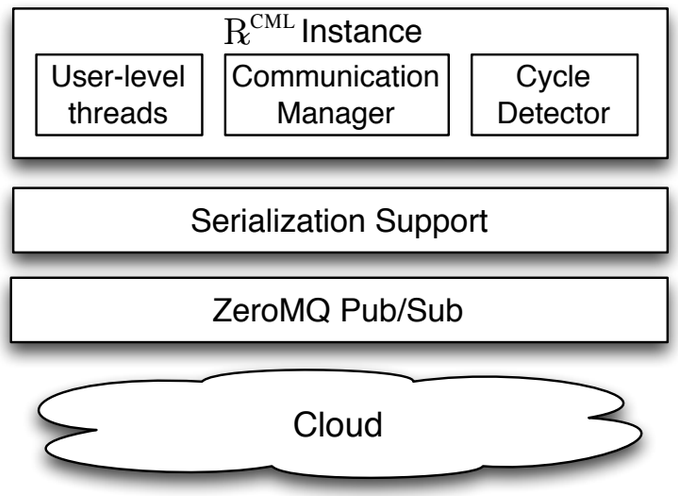


Not a well-formed execution

Implementation: Overview

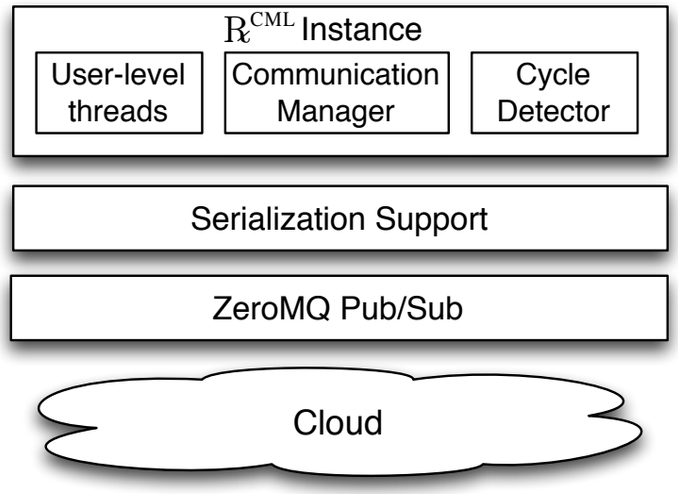
Implementation: Overview

- Rx-CML: **Rela**Xed **CML**
  - ★ MultiMLton with distribution support
  - ★ Rx-CML application = {Instances}
  - ★ Supports full CML
  - ★ Built-in serialization (immutable values and function closures)
  - ★ Transport layer is ZeroMQ



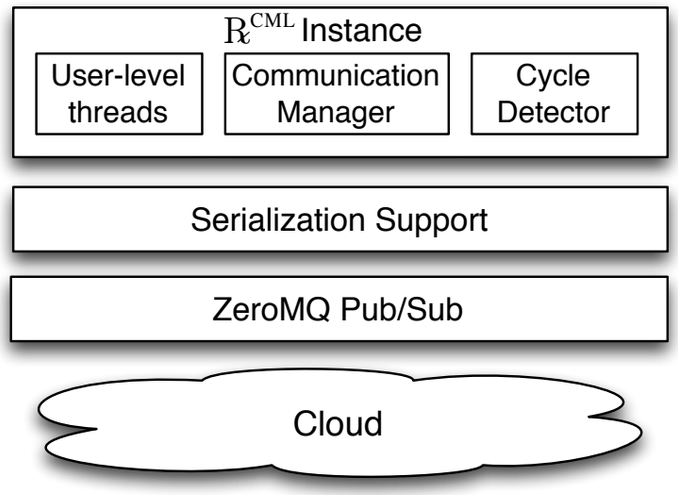
Implementation: Overview

- Rx-CML: **Rela**Xed **CML**
  - ★ MultiMLton with distribution support
  - ★ Rx-CML application = {Instances}
  - ★ Supports full CML
  - ★ Built-in serialization (immutable values and function closures)
  - ★ Transport layer is ZeroMQ
- Check the integrity of the speculative actions on-the-fly



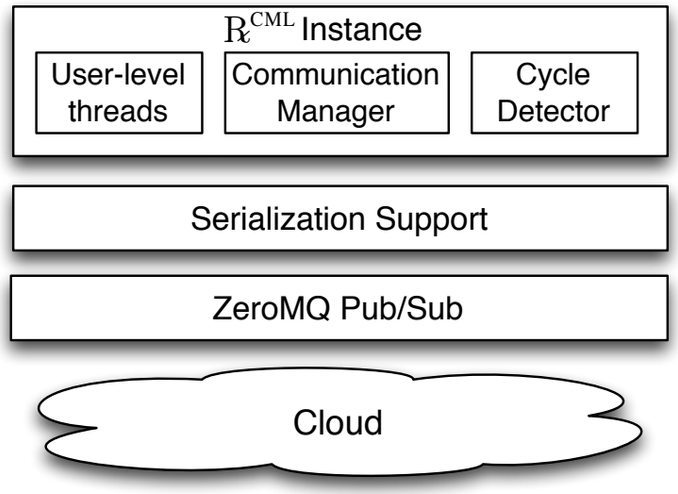
Implementation: Overview

- Rx-CML: **R**ela**X**ed **CML**
  - ★ MultiMLton with distribution support
  - ★ Rx-CML application = {Instances}
  - ★ Supports full CML
  - ★ Built-in serialization (immutable values and function closures)
  - ★ Transport layer is ZeroMQ
- Check the integrity of the speculative actions on-the-fly
  - ★ Build a dependence graph that captures happens-before relation



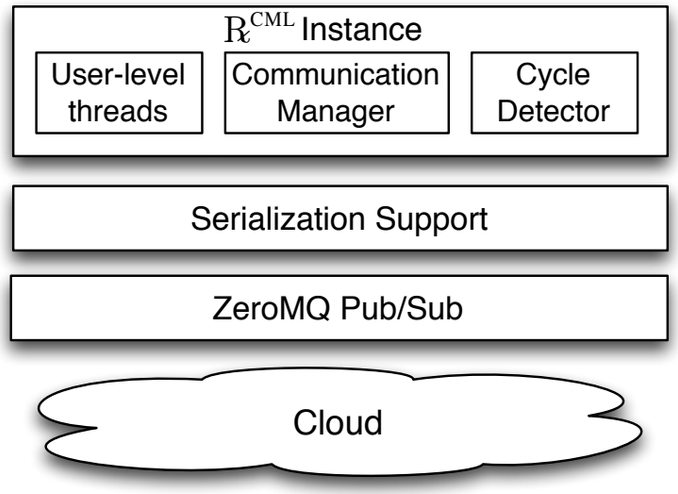
Implementation: Overview

- Rx-CML: **R**ela**X**ed **CML**
  - ★ MultiMLton with distribution support
  - ★ Rx-CML application = {Instances}
  - ★ Supports full CML
  - ★ Built-in serialization (immutable values and function closures)
  - ★ Transport layer is ZeroMQ
- Check the integrity of the speculative actions on-the-fly
  - ★ Build a dependence graph that captures happens-before relation
    - ✦ Same structure as an axiomatic execution



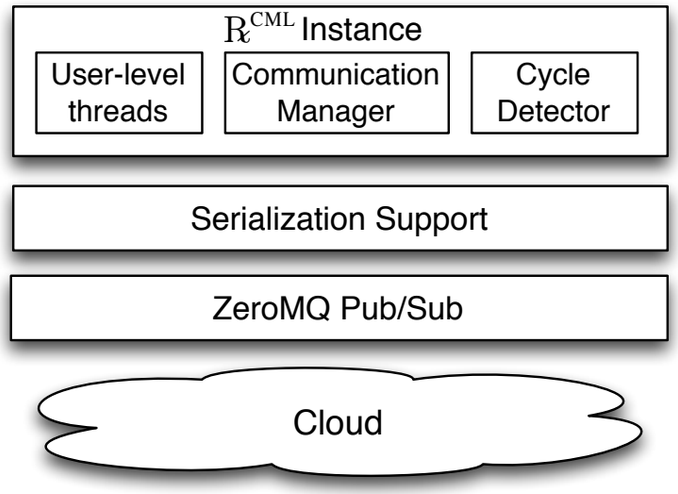
Implementation: Overview

- Rx-CML: **Rela**Xed **CML**
  - ★ MultiMLton with distribution support
  - ★ Rx-CML application = {Instances}
  - ★ Supports full CML
  - ★ Built-in serialization (immutable values and function closures)
  - ★ Transport layer is ZeroMQ
- Check the integrity of the speculative actions on-the-fly
  - ★ Build a dependence graph that captures happens-before relation
    - ✦ Same structure as an axiomatic execution
  - ★ Automatically check dependence graph integrity before an observable action (ref cell accesses, system calls, FFI, etc)



Implementation: Overview

- Rx-CML: **Rela**Xed **CML**
  - ★ MultiMLton with distribution support
  - ★ Rx-CML application = {Instances}
  - ★ Supports full CML
  - ★ Built-in serialization (immutable values and function closures)
  - ★ Transport layer is ZeroMQ
- Check the integrity of the speculative actions on-the-fly
  - ★ Build a dependence graph that captures happens-before relation
    - ✦ Same structure as an axiomatic execution
  - ★ Automatically check dependence graph integrity before an observable action (ref cell accesses, system calls, FFI, etc)
  - ★ Roll-back ill-formed executions, re-execute non-speculatively



Channel Consistency (1)

Channel Consistency (1)

- Single consistent channel image across all instances without coherence?

Channel Consistency (1)

- Single consistent channel image across all instances without coherence?
  - ★ Communication manager thread @ every instance

Channel Consistency (1)

- Single consistent channel image across all instances without coherence?
  - ★ Communication manager thread @ every instance
  - ★ Maintains a replica of CML channel

Channel Consistency (1)

- Single consistent channel image across all instances without coherence?
  - ★ Communication manager thread @ every instance
  - ★ Maintains a replica of CML channel
  - ★ Exploit speculative execution maintaining consistency

Channel Consistency (1)

- Single consistent channel image across all instances without coherence?
  - ★ Communication manager thread @ every instance
  - ★ Maintains a replica of CML channel
  - ★ Exploit speculative execution maintaining consistency

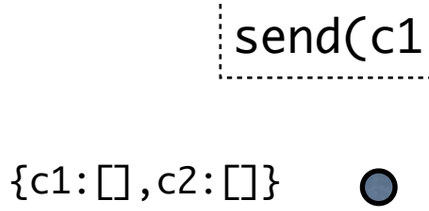
Instance 1

```
send(c1,0);
```

Instance 2

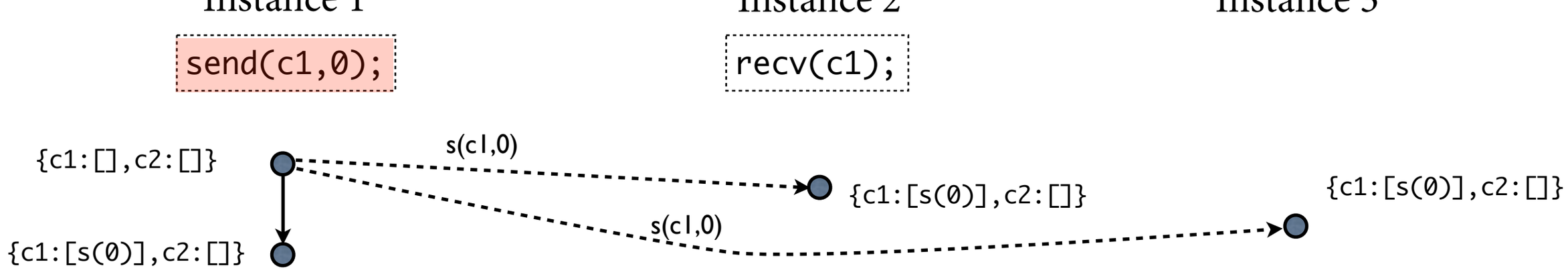
```
recv(c1);
```

Instance 3



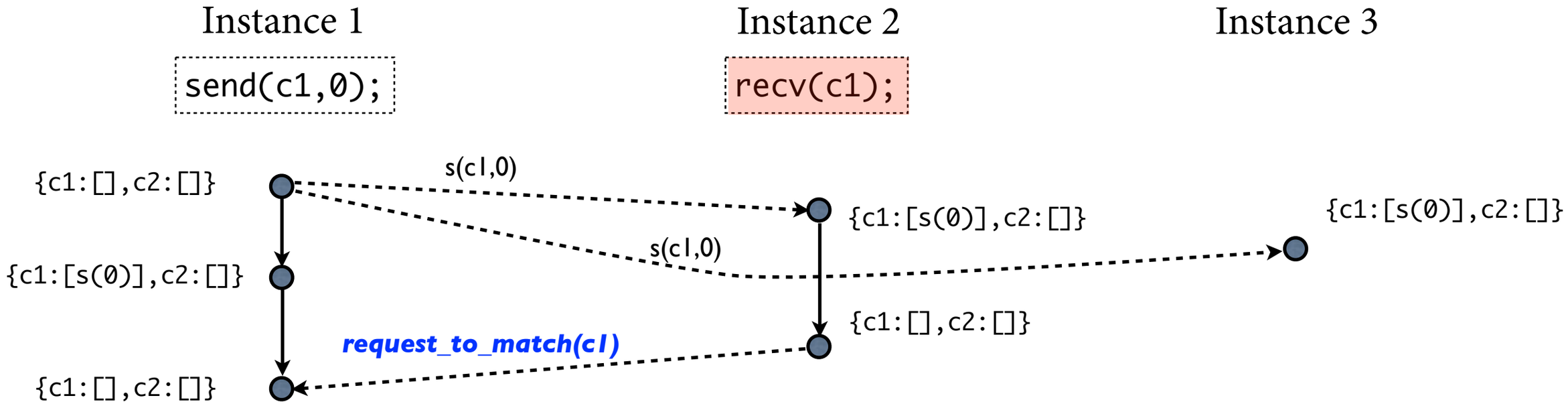
Channel Consistency (1)

- Single consistent channel image across all instances without coherence?
  - ★ Communication manager thread @ every instance
  - ★ Maintains a replica of CML channel
  - ★ Exploit speculative execution maintaining consistency



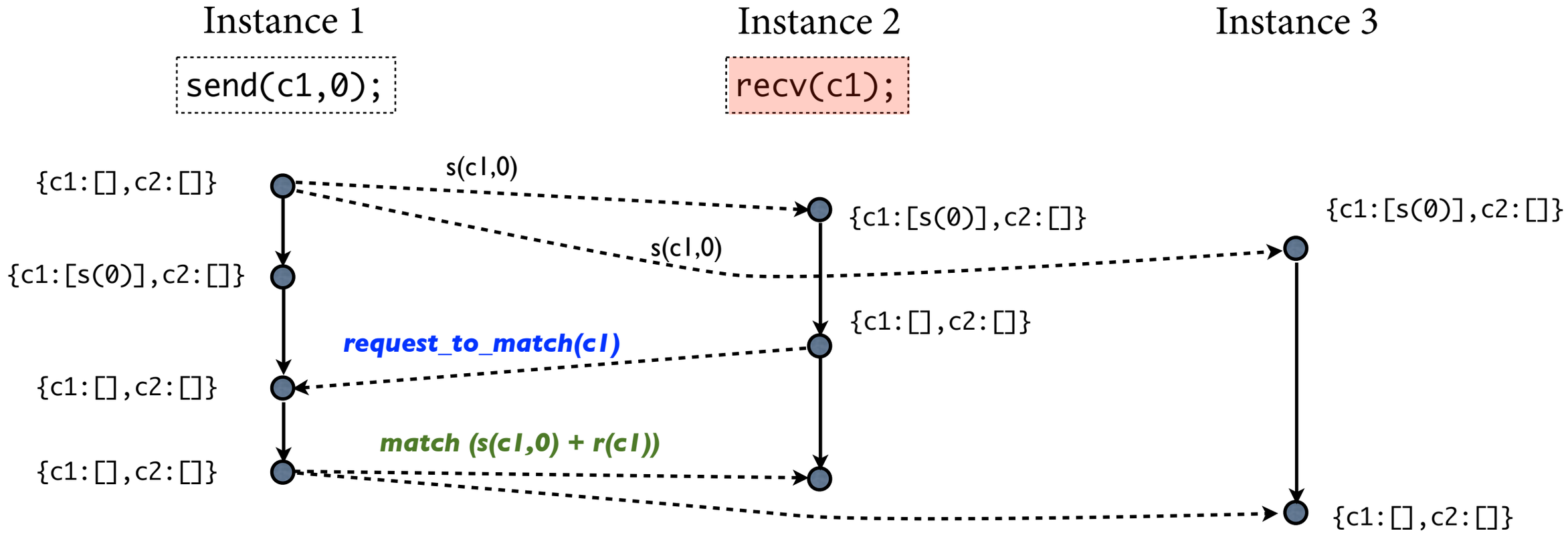
Channel Consistency (1)

- Single consistent channel image across all instances without coherence?
  - ★ Communication manager thread @ every instance
  - ★ Maintains a replica of CML channel
  - ★ Exploit speculative execution maintaining consistency



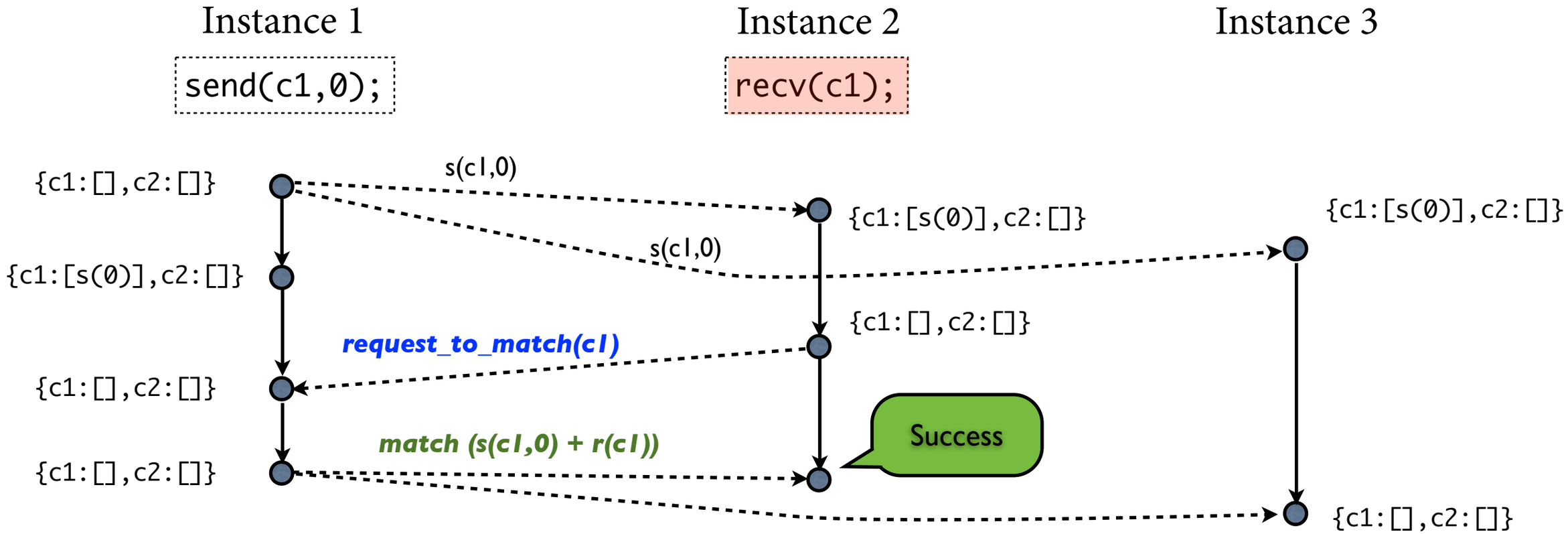
Channel Consistency (1)

- Single consistent channel image across all instances without coherence?
  - ★ Communication manager thread @ every instance
  - ★ Maintains a replica of CML channel
  - ★ Exploit speculative execution maintaining consistency



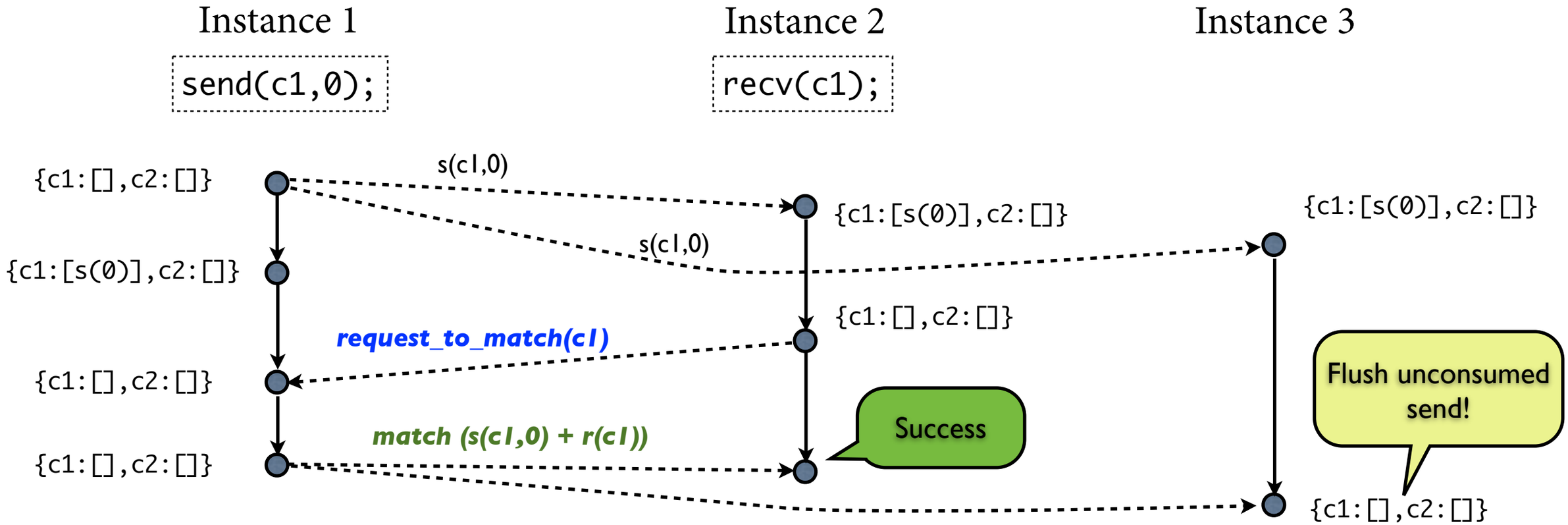
Channel Consistency (1)

- Single consistent channel image across all instances without coherence?
  - ★ Communication manager thread @ every instance
  - ★ Maintains a replica of CML channel
  - ★ Exploit speculative execution maintaining consistency



Channel Consistency (1)

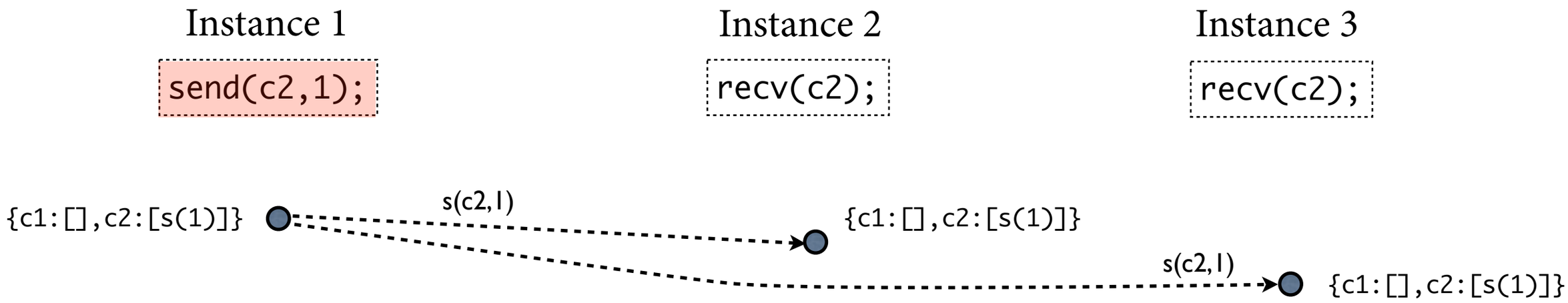
- Single consistent channel image across all instances without coherence?
  - ★ Communication manager thread @ every instance
  - ★ Maintains a replica of CML channel
  - ★ Exploit speculative execution maintaining consistency



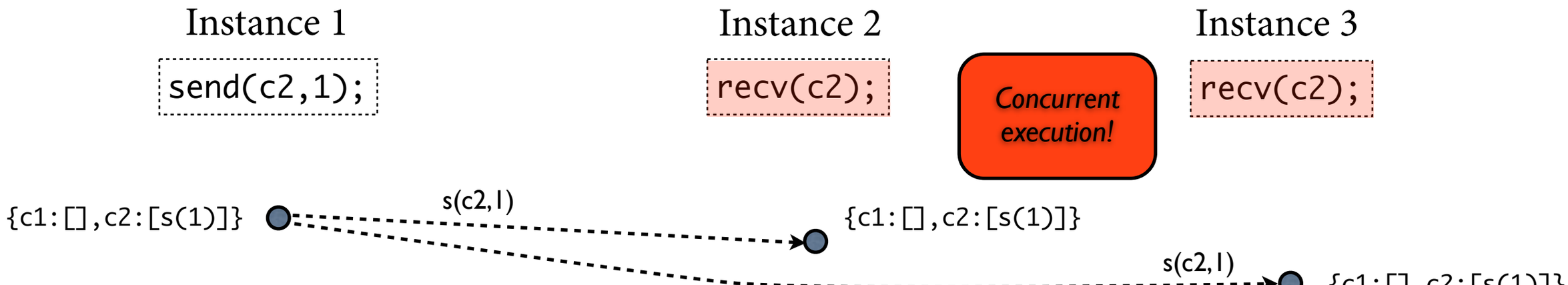
Channel Consistency (2)

Instance 1	Instance 2	Instance 3
`send(c2,1);`	`recv(c2);`	`recv(c2);`

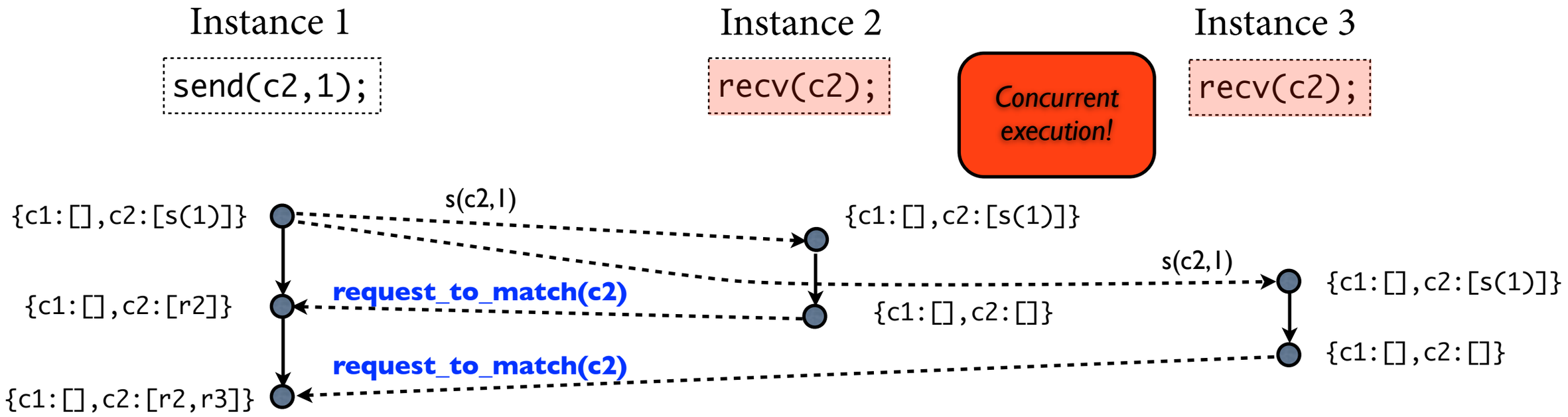
Channel Consistency (2)



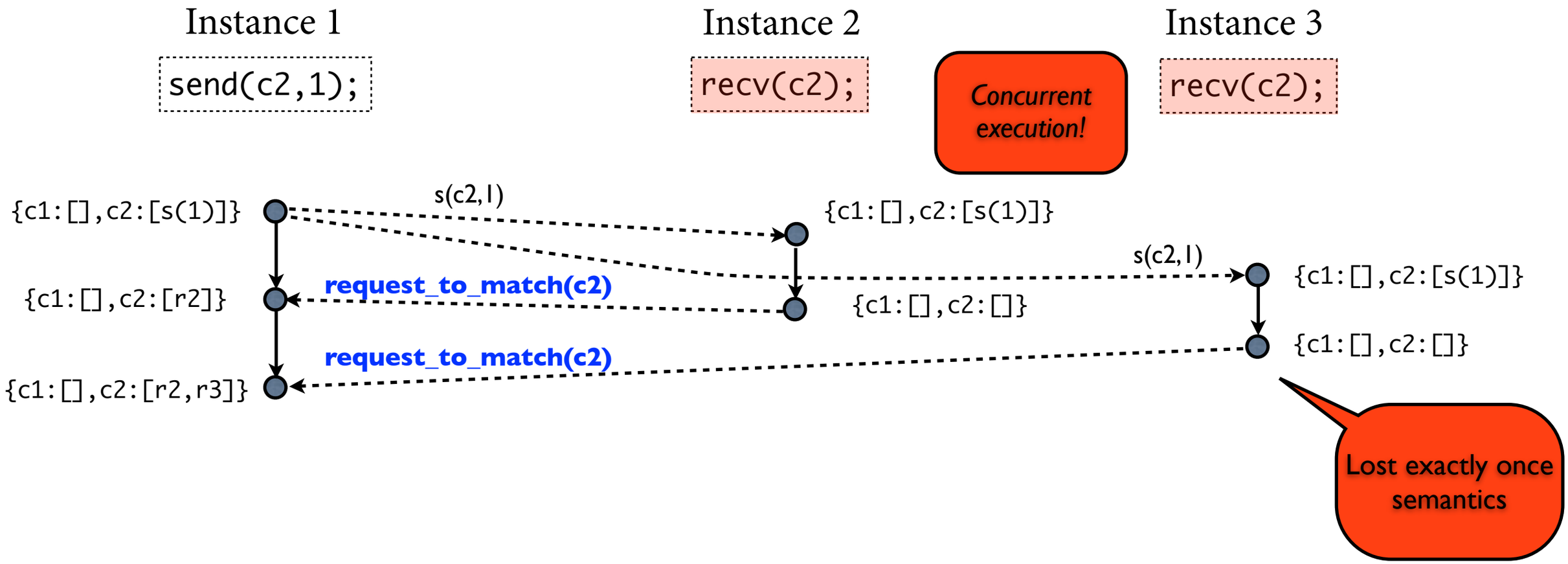
Channel Consistency (2)



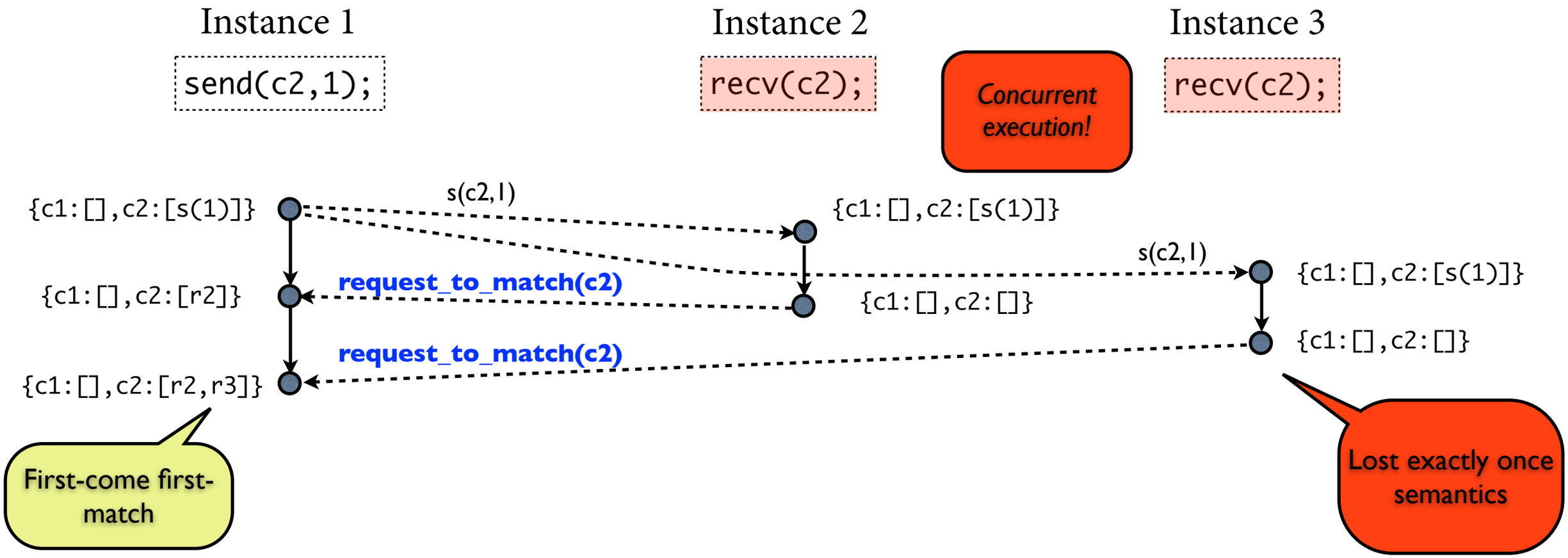
Channel Consistency (2)



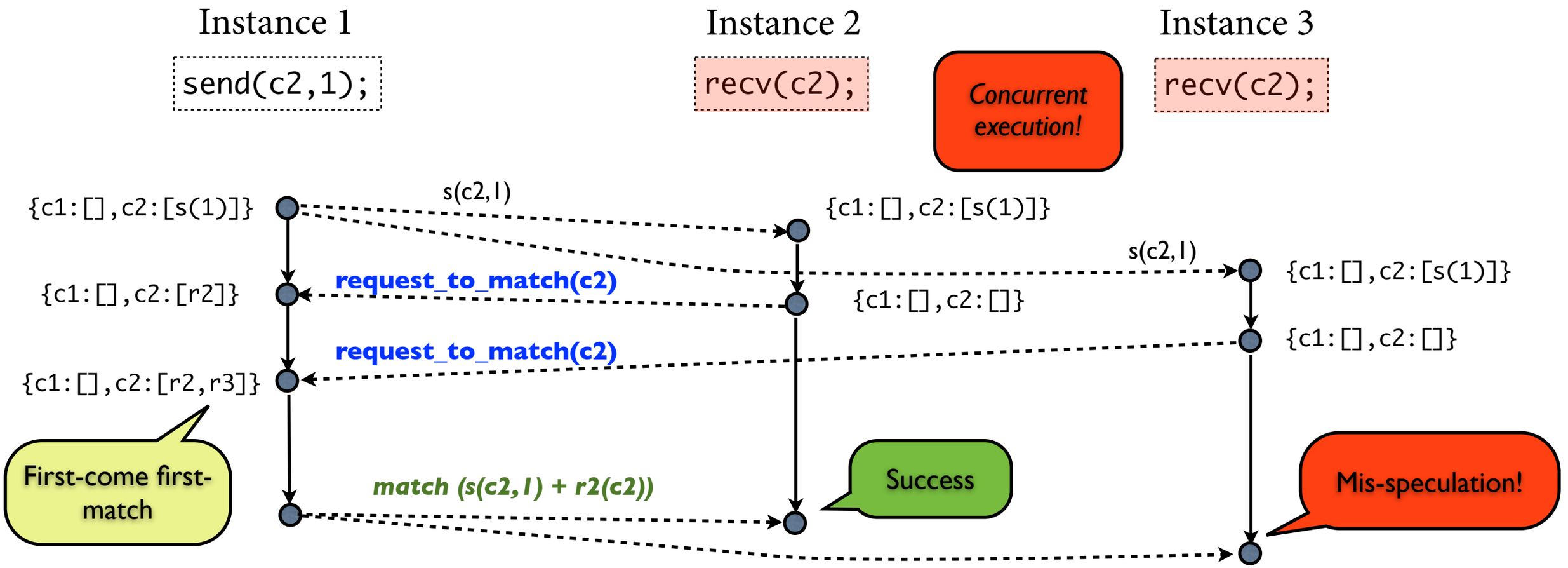
Channel Consistency (2)



Channel Consistency (2)



Channel Consistency (2)



Speculative Execution

Speculative Execution

- Consistent, replicated dependence graph @ each instance
  - ★ *Snoop on match messages* from communication manager
  - ★ Broadcast thread spawn and thread join messages

Speculative Execution

- Consistent, replicated dependence graph @ each instance
  - ★ *Snoop on match messages* from communication manager
  - ★ Broadcast thread spawn and thread join messages
- Well-formedness check automatically before observable actions
  - ★ Check is local to the instance!
  - ★ GC dependence graph on successful well-formedness check

Speculative Execution

- Consistent, replicated dependence graph @ each instance
  - ★ *Snoop on match messages* from communication manager
  - ★ Broadcast thread spawn and thread join messages
- Well-formedness check automatically before observable actions
  - ★ Check is local to the instance!
  - ★ GC dependence graph on successful well-formedness check
- Automatic checkpointing
  - ★ 1 continuation per thread
  - ★ *Uncoordinated!* - thread local - does not require barriers

Speculative Execution

- Consistent, replicated dependence graph @ each instance
  - ★ *Snoop on match messages* from communication manager
  - ★ Broadcast thread spawn and thread join messages
- Well-formedness check automatically before observable actions
  - ★ Check is local to the instance!
  - ★ GC dependence graph on successful well-formedness check
- Automatic checkpointing
  - ★ 1 continuation per thread
  - ★ *Uncoordinated!* - thread local - does not require barriers
- Remediation
  - ★ *Uncoordinated!* - Transitively inform each mis-speculated thread to rollback
  - ★ *Check-point (Continuation) + Log-based (Dependence graph)* recovery
  - ★ Rollback to last checkpoint, replay correct speculative actions
  - ★ Continues non-speculatively until next observable action = Progress

Results

Results

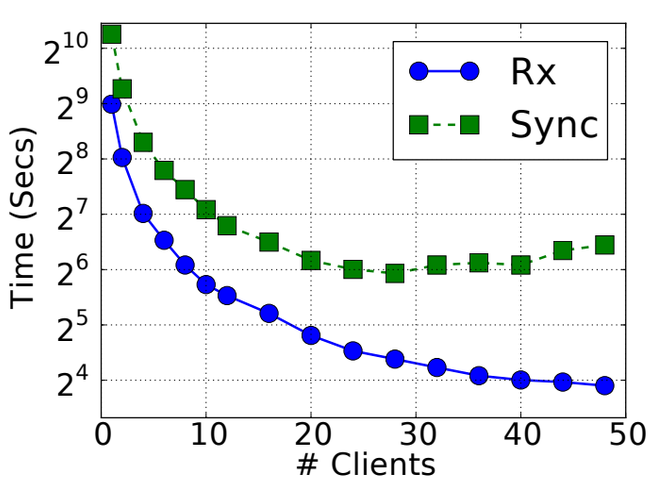
- Optimistic OLTP
  - ★ Distributed version of STAMP Vacation benchmark
  - ★ Database split into 64 shards, with concurrent transaction requests from geo-distributed clients
  - ★ Uses explicit lock servers -> Rx-CML executes transactions optimistically

Results

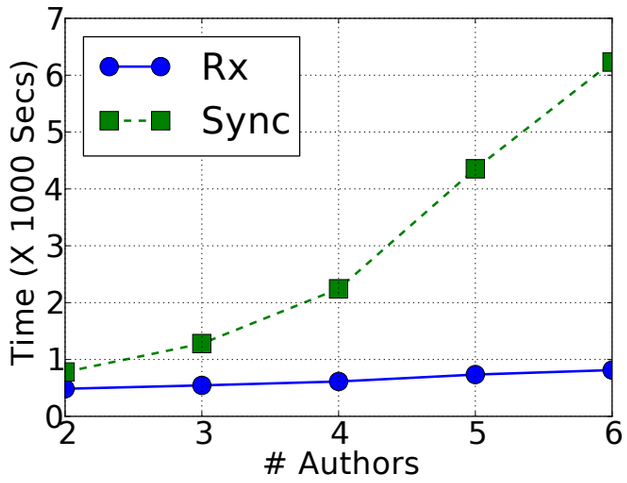
- Optimistic OLTP
  - ★ Distributed version of STAMP Vacation benchmark
  - ★ Database split into 64 shards, with concurrent transaction requests from geo-distributed clients
  - ★ Uses explicit lock servers -> Rx-CML executes transactions optimistically
- P2P Collaborative editing
  - ★ Simulates concurrent document editing (operational transformation)
  - ★ Total order broadcast - built out of synchronous events + choice combinator.

Results

- Optimistic OLTP
  - ★ Distributed version of STAMP Vacation benchmark
  - ★ Database split into 64 shards, with concurrent transaction requests from geo-distributed clients
  - ★ Uses explicit lock servers -> Rx-CML executes transactions optimistically
- P2P Collaborative editing
  - ★ Simulates concurrent document editing (operational transformation)
  - ★ Total order broadcast - built out of synchronous events + choice combinator.



OLTP



Collaborative Editing

- Rx-CML was 5.8X to 7.6X faster than the synchronous version
  - ★ 9-17% of communications were mis-speculated.

Conclusion

Conclusion

- Composable synchronous events vs. the Cloud

PURDUE UNIVERSITY

MultiMLton

Conclusion

- Composable synchronous events vs. the Cloud
- Rx-CML (Relaxed CML)
  - ★ reason synchronously, but implement asynchronously
  - ★ retain simplicity and composability, but gain performance
  - ★ optimistic concurrency control for CML

Conclusion

- Composable synchronous events vs. the Cloud
- Rx-CML (Relaxed CML)
  - ★ reason synchronously, but implement asynchronously
  - ★ retain simplicity and composability, but gain performance
  - ★ optimistic concurrency control for CML
- Distributed implementation of MultiMLton
  - ★ Case studies demonstrate effectiveness of the approach

Conclusion

- Composable synchronous events vs. the Cloud
- Rx-CML (Relaxed CML)
  - ★ reason synchronously, but implement asynchronously
  - ★ retain simplicity and composability, but gain performance
  - ★ optimistic concurrency control for CML
- Distributed implementation of MultiMLton
  - ★ Case studies demonstrate effectiveness of the approach
- Future Work - Fault tolerance
  - ★ Make checkpoints and dependence graph resilient
  - ★ Treat failures as mis-speculations -> rollback to last saved checkpoint

Questions?

http://multimlton.cs.purdue.edu