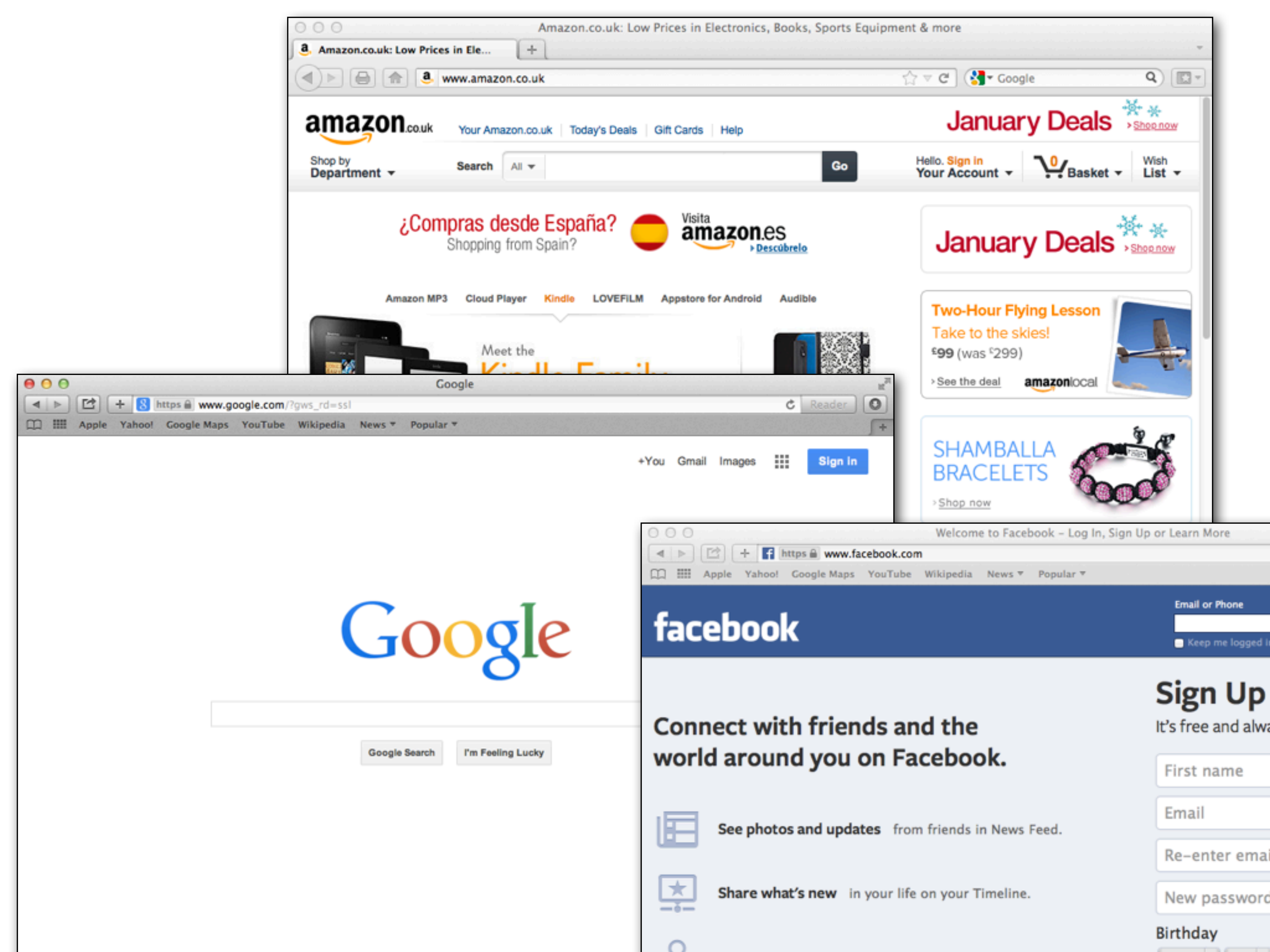
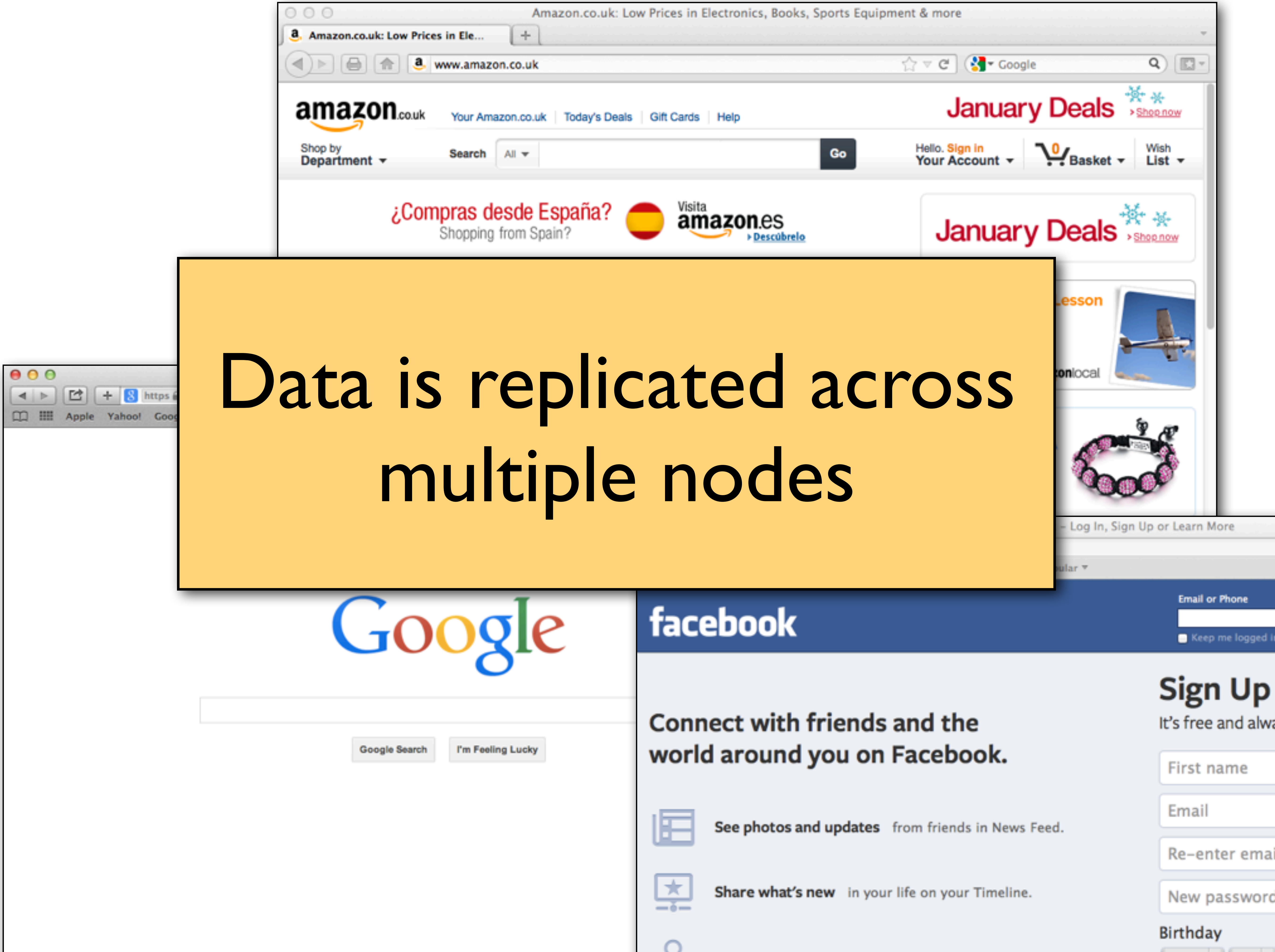
Composite Replicated Data Types

Alexey Gotsman
IMDEA Software Institute, Madrid, Spain

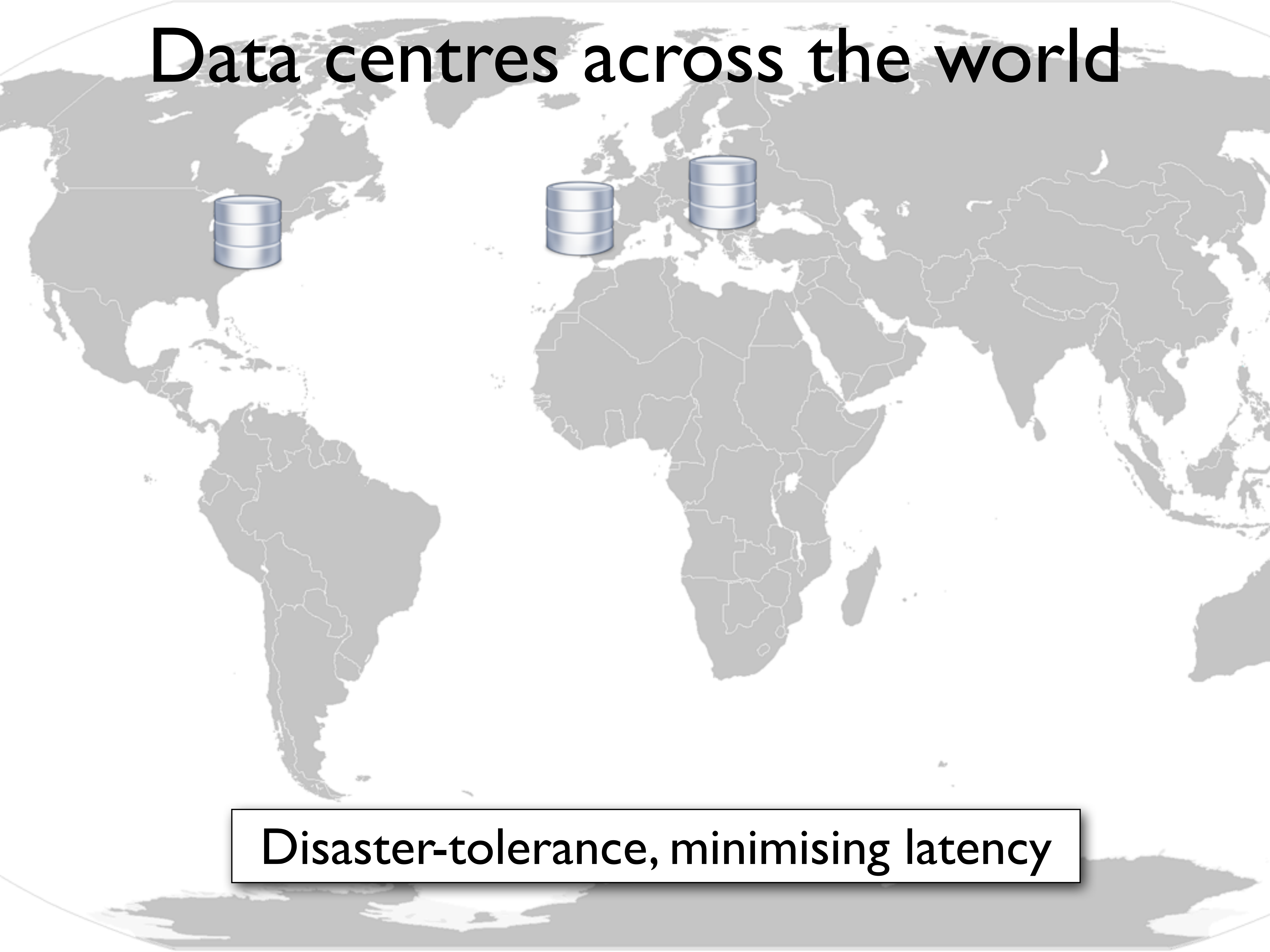
Joint work with Hongseok Yang (Oxford)


Amazon.co.uk: Low Prices in Electronics, Books, Sports Equipment & more
www.amazon.co.uk
Your Amazon.co.uk | Today's Deals | Gift Cards | Help
January Deals
Shop by Department
Search
Go
Hello. Sign in Your Account
Basket
Wish List
¿Compras desde España?
Shopping from Spain?
Visita amazon.es
Descúbrelo
Amazon MP3 Cloud Player Kindle LOVEFiLM Appstore for Android Audible
Meet the
Two-Hour Flying Lesson
Take to the skies!
£99 (was £299)
See the deal
SHAMBALLA BRACELETS
Shop now
Google
www.google.com/?gws_rd=ssl
Apple Yahoo! Google Maps YouTube Wikipedia News Popular
+You Gmail Images
Sign in
Google
Google Search
I'm Feeling Lucky
Welcome to Facebook – Log In, Sign Up or Learn More
www.facebook.com
facebook
Email or Phone
Sign Up
Connect with friends and the world around you on Facebook.
See photos and updates from friends in News Feed.
Share what's new in your life on your Timeline.
First name
Email
Birthday

Data is replicated across multiple nodes

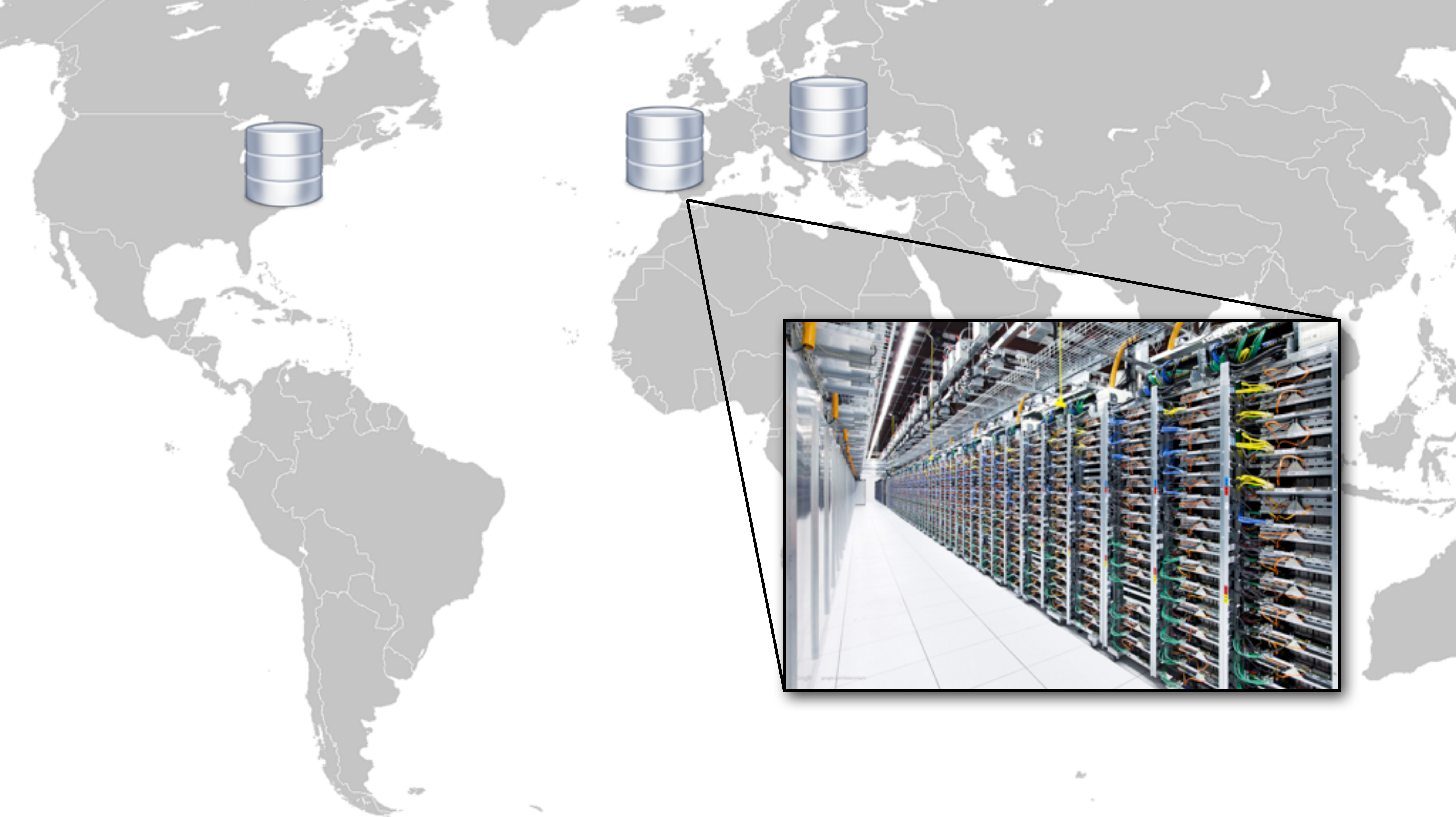


Data centres across the world



Disaster-tolerance, minimising latency

With thousands of machines inside

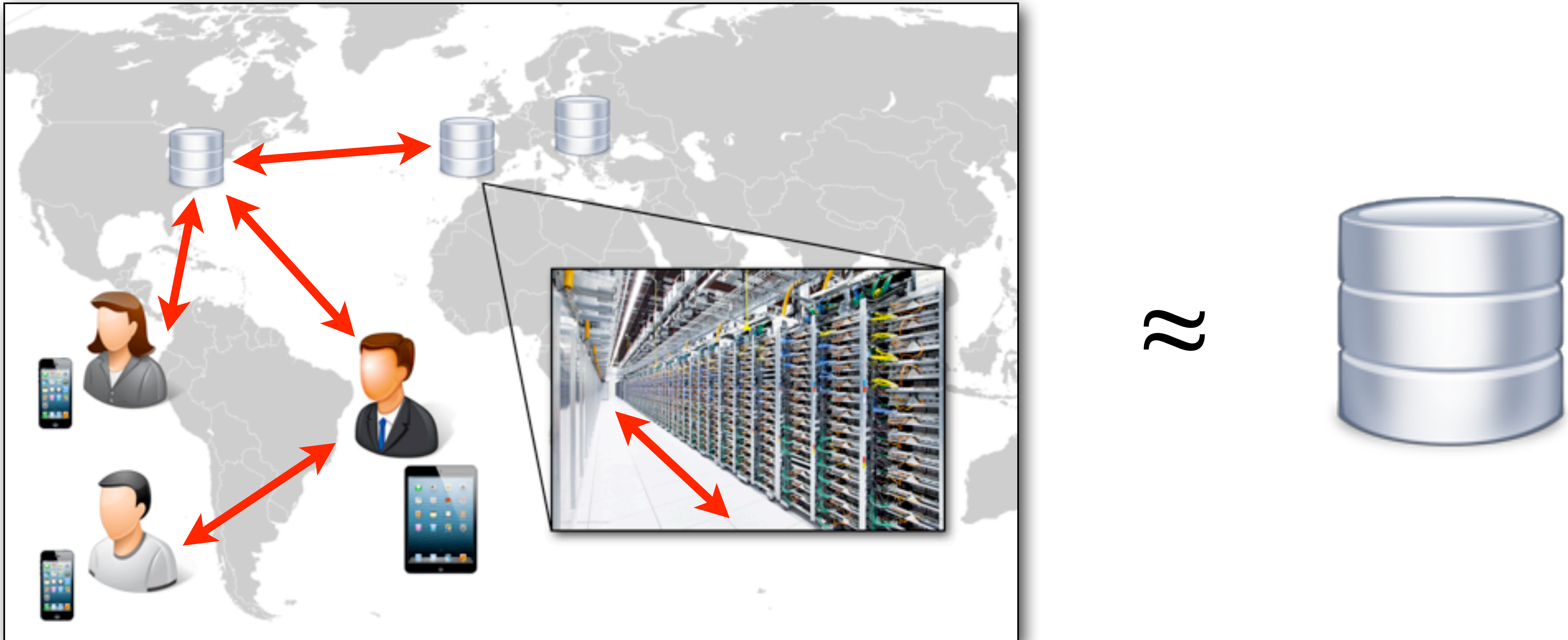


Load-balancing, fault-tolerance

Replicas on mobile devices

Offline use

- Serialisability: the system behaves like a serial processor of transactions on a centralised database



- Serialisability: the system behaves like a serial processor of transactions on a centralised database
- Requires synchronisation: contact other replicas when processing a request
- Expensive or impossible

Eventually consistent databases

friends[*Alice*].add(*Bob*)
friends[*Bob*].add(*Alice*)

→ …

No synchronisation: update your replica now, propagate to others later

Eventually consistent databases

friends[*Alice*].add(*Bob*)
friends[*Bob*].add(*Alice*)

→

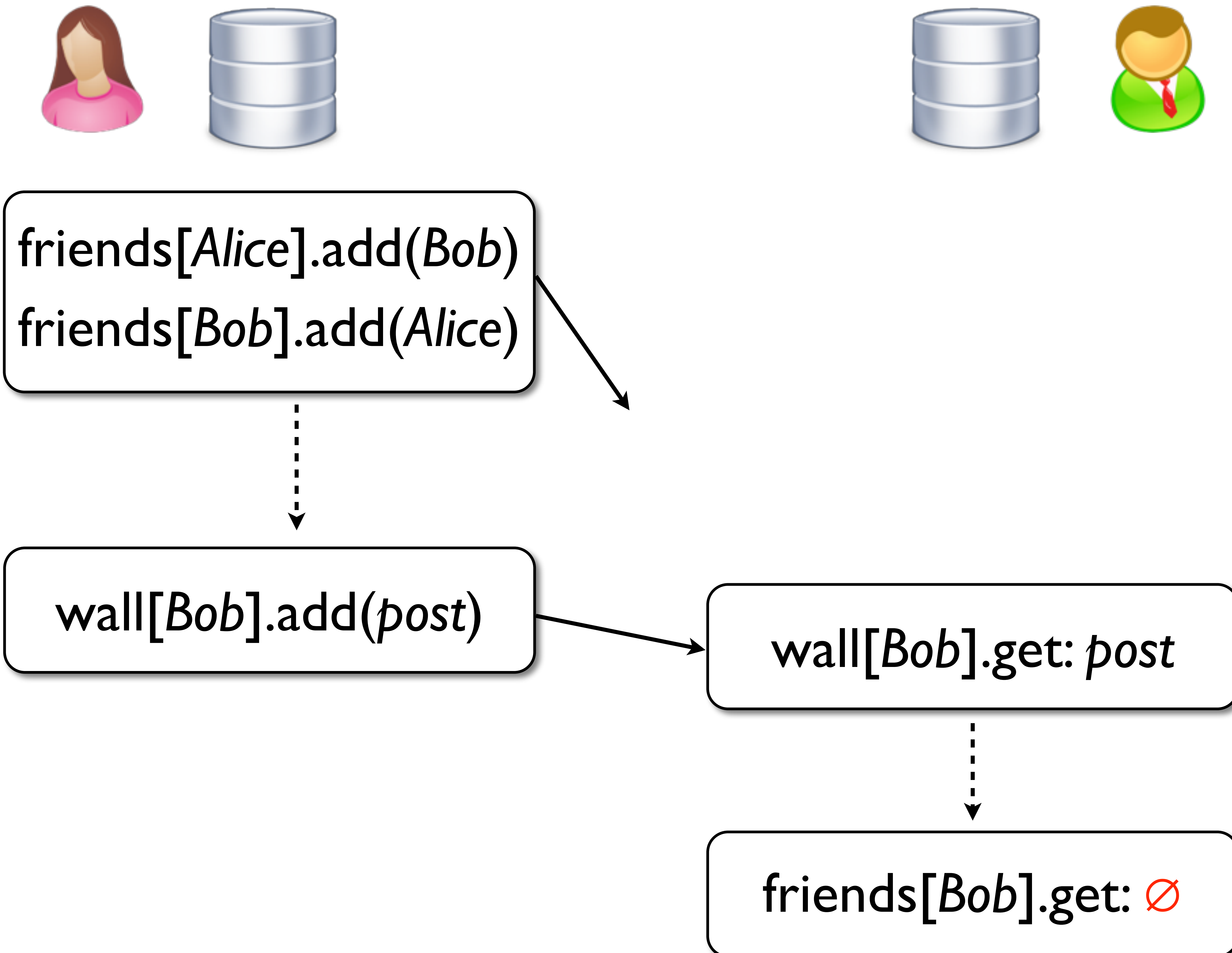
friends[*Alice*].get: {*Bob*}
friends[*Bob*].get: {*Alice*}

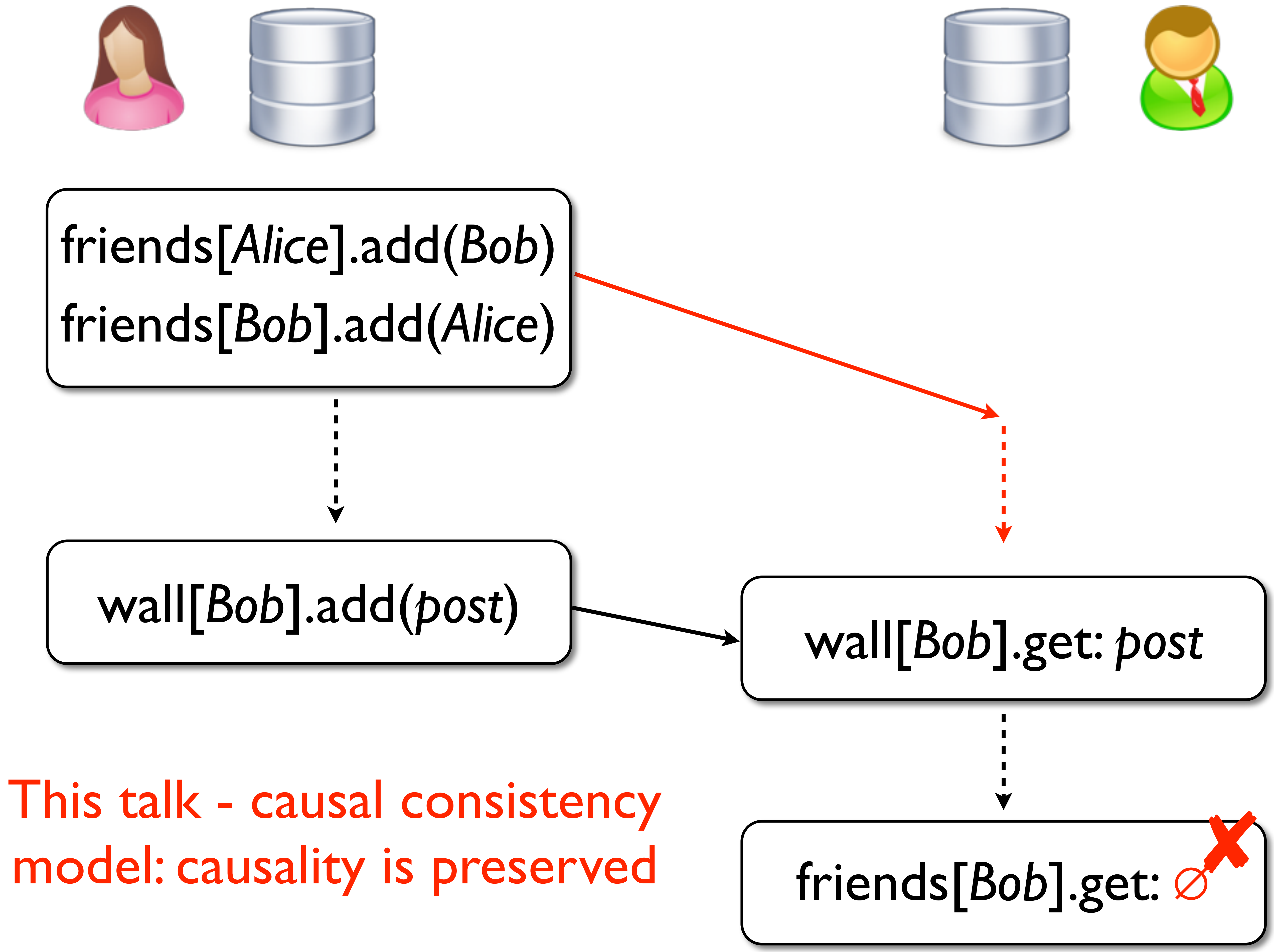
- All updates by the transaction delivered together
- Can preserve invariants, but isn't serialisability: asynchronous communication leads to anomalies

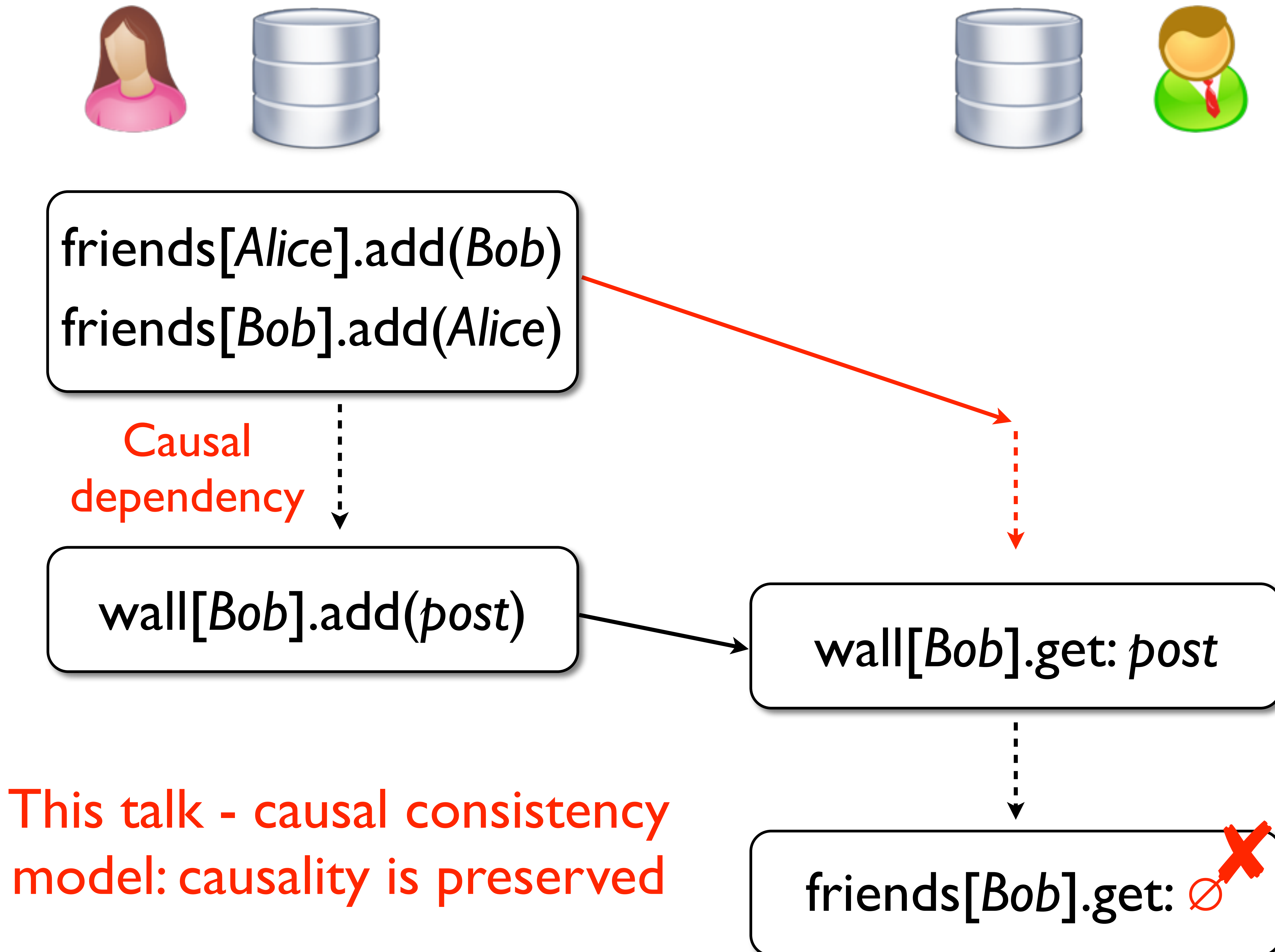
friends[*Alice*].add(*Bob*)
friends[*Bob*].add(*Alice*)

↓

wall[*Bob*].add(*post*)


friends[*Alice*].add(*Bob*)
friends[*Bob*].add(*Alice*)
wall[*Bob*].add(*post*)
wall[*Bob*].get: *post*
friends[*Bob*].get: ∅


friends[Alice].add(Bob)
friends[Bob].add(Alice)
wall[Bob].add(post)
wall[Bob].get: post
friends[Bob].get: ∅
This talk - causal consistency
model: causality is preserved


friends[*Alice*].add(*Bob*)
friends[*Bob*].add(*Alice*)
Causal dependency
wall[*Bob*].add(*post*)
wall[*Bob*].get: *post*
friends[*Bob*].get: ∅
This talk - causal consistency model: causality is preserved

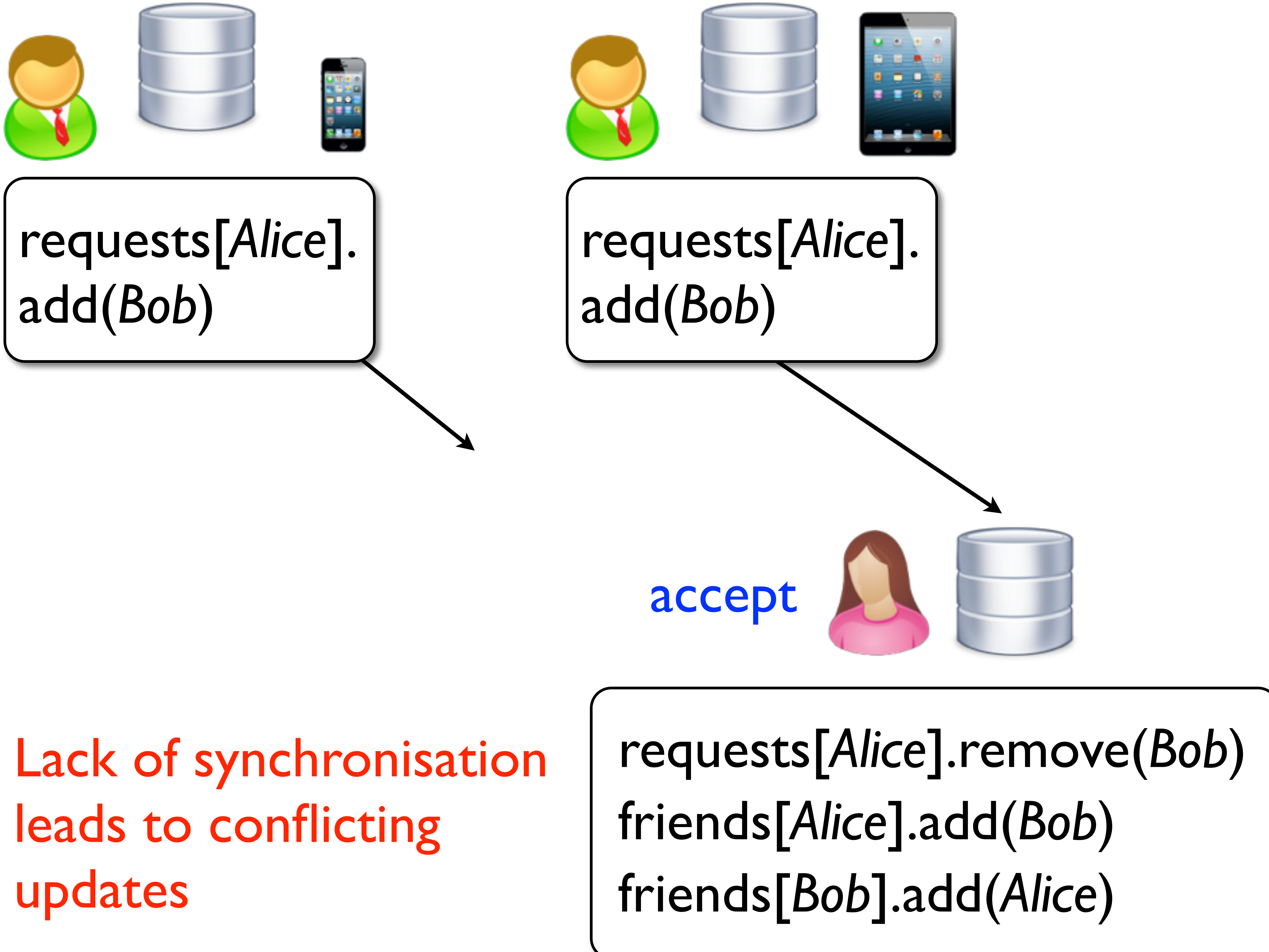
requests[*Alice*].
add(*Bob*)

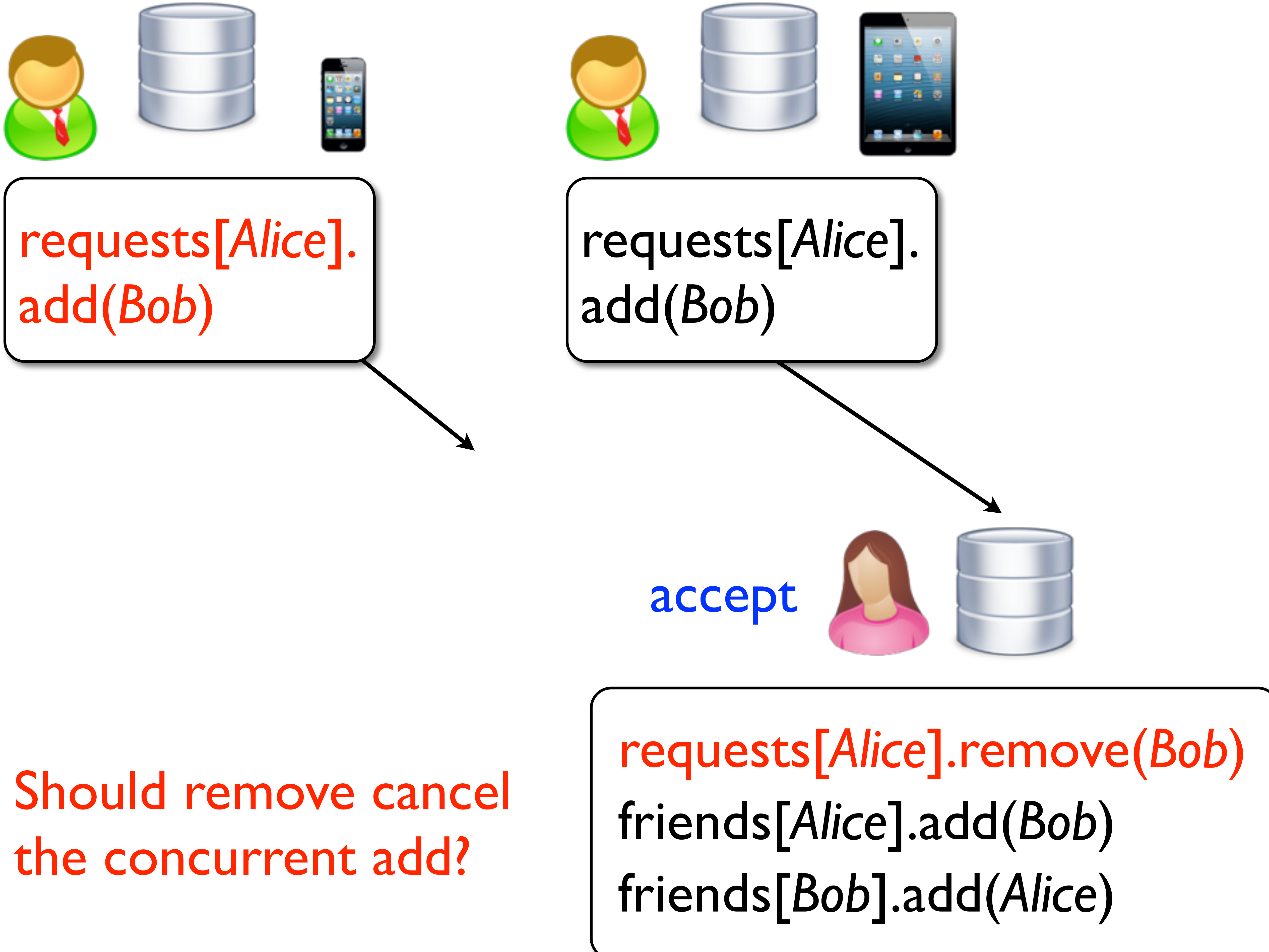
Lack of synchronisation
leads to conflicting
updates

requests[*Alice*].
add(*Bob*)

requests[*Alice*].
add(*Bob*)

Lack of synchronisation leads to conflicting updates


requests[Alice].
add(Bob)
requests[Alice].
add(Bob)
accept
requests[Alice].remove(Bob)
friends[Alice].add(Bob)
friends[Bob].add(Alice)
Lack of synchronisation
leads to conflicting
updates



Should remove cancel the concurrent add?

requests[*Alice*].add(*Bob*)

Conflict-resolution policies:

Remove wins: requests[*Alice*] = ∅

Add wins: requests[*Alice*] = {*Bob*}

accept

requests[*Alice*].remove(*Bob*)
friends[*Alice*].add(*Bob*)
friends[*Bob*].add(*Alice*)

Should remove cancel the concurrent add?

requests[*Alice*].add(*Bob*)

Conflict-resolution policies:

Remove wins: requests[*Alice*] = ∅

Add wins: requests[*Alice*] = {*Bob*}

accept

requests[*Alice*].remove(*Bob*)
friends[*Alice*].add(*Bob*)
friends[*Bob*].add(*Alice*)

Should remove cancel the concurrent add?

Conflict-resolution policies:

Remove wins: requests[*Alice*] = ∅

Add wins: requests[*Alice*] = {*Bob*}

- Encapsulated in replicated data types (aka CRDTs) [Shapiro+ 2011]
- Object ➜ Type ➜ Conflict-resolution policy
- Remove-wins set, add-wins set, counters, registers...

The brave new world of eventual consistency

Features to maintain correctness despite weak consistency:

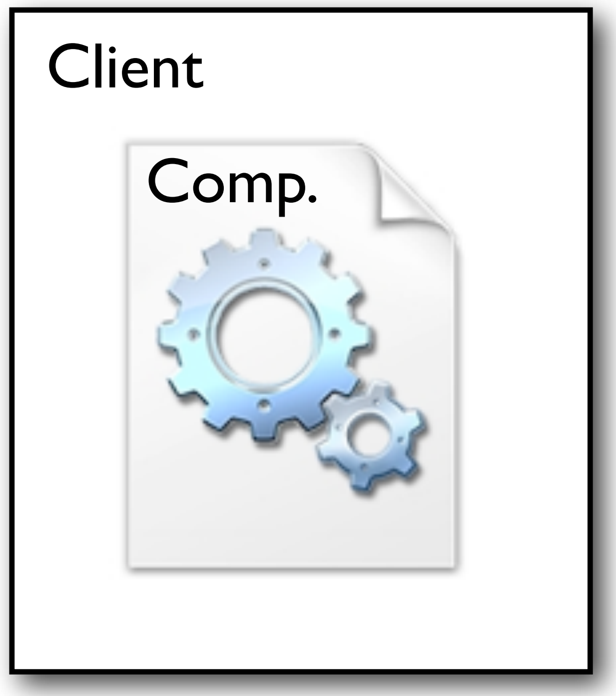
- Consistency models restricting anomalies: causal consistency
- Programming concepts: replicated data types, transactions

Subtle semantics ➜ programming is difficult

Goal

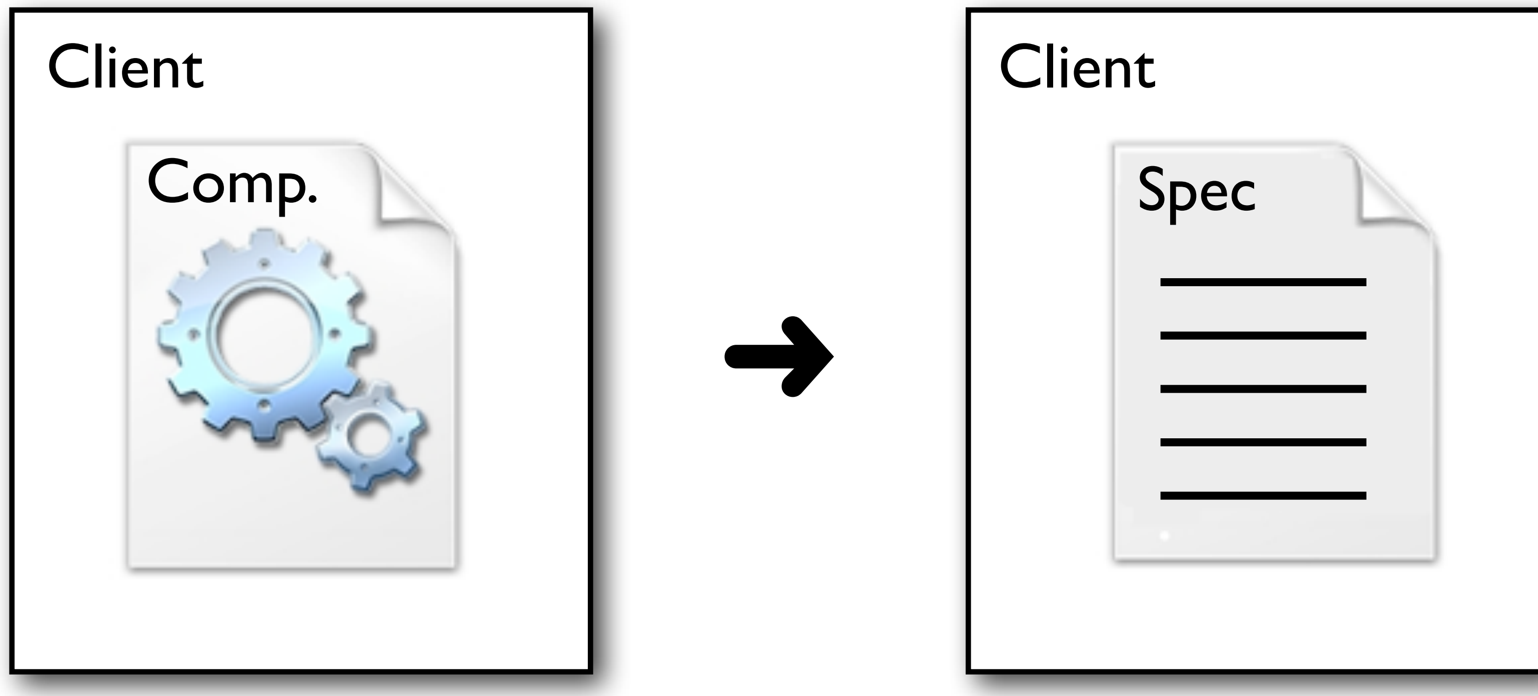
- Want to help programmers: specification, verification, programming models
- Need a better theoretical understanding of programming on eventual consistency
- Groundwork: formal semantics specification [POPL'14]
  - Executions represented using structures of events and partial orders
  - Replicated data type specifications

This talk: modularity



- Programmers encapsulate functionality into components
- Use a component without knowing its implementation

This talk: modularity



- Programmers encapsulate functionality into components
- Use a component without knowing its implementation
- Want to abstract from component internals when reasoning about its client ➜ replace it by a spec

Composite replicated data types

≈ ADT, module

```
datatype SocialGraph {
 RemoveWinsSet friends[];
 RemoveWinsSet requests[];

 tx accept(Bob→Alice) {
  ...
  requests[Alice].
              remove(Bob);
  friends[Alice].add(Bob);
  friends[Bob].add(Alice);
  ...
 }
}
```

Composite replicated data types

≈ ADT, module

```
datatype SocialGraph {
 RemoveWinsSet friends[];
 RemoveWinsSet requests[];

 tx accept(Bob→Alice) {
  ...
  requests[Alice].
              remove(Bob);
  friends[Alice].add(Bob);
  friends[Bob].add(Alice);
  ...
 }
}
```

- Objects of replicated data types

Composite replicated data types

≈ ADT, module

```
datatype SocialGraph {
 RemoveWinsSet friends[];
 RemoveWinsSet requests[];

 tx accept(Bob→Alice) {
  ...
  requests[Alice].
             remove(Bob);
  friends[Alice].add(Bob);
  friends[Bob].add(Alice);
  ...
 }
}
```

- Objects of replicated data types
- Information hiding: no direct access to the objects
- Methods implemented by causally consistent transactions

Composite replicated data types

≈ ADT, module

```
datatype SocialGraph {
 RemoveWinsSet friends[];
 RemoveWinsSet requests[];

 tx accept(Bob→Alice) {
  ...
  requests[Alice].
             remove(Bob);
  friends[Alice].add(Bob);
  friends[Bob].add(Alice);
  ...
 }
}
```

- Replicated data types ensure conflict resolution
- Transactions maintain integrity invariants
- Want to abstract from the implementation when reasoning about client programs

```
datatype SocialGraph {
 RemoveWinsSet friends[];
 RemoveWinsSet requests[];

 tx accept(Bob → Alice) {
  ...
  requests[Alice].remove(Bob);
  friends[Alice].add(Bob);
  friends[Bob].add(Alice);
  ...
 }
}
```

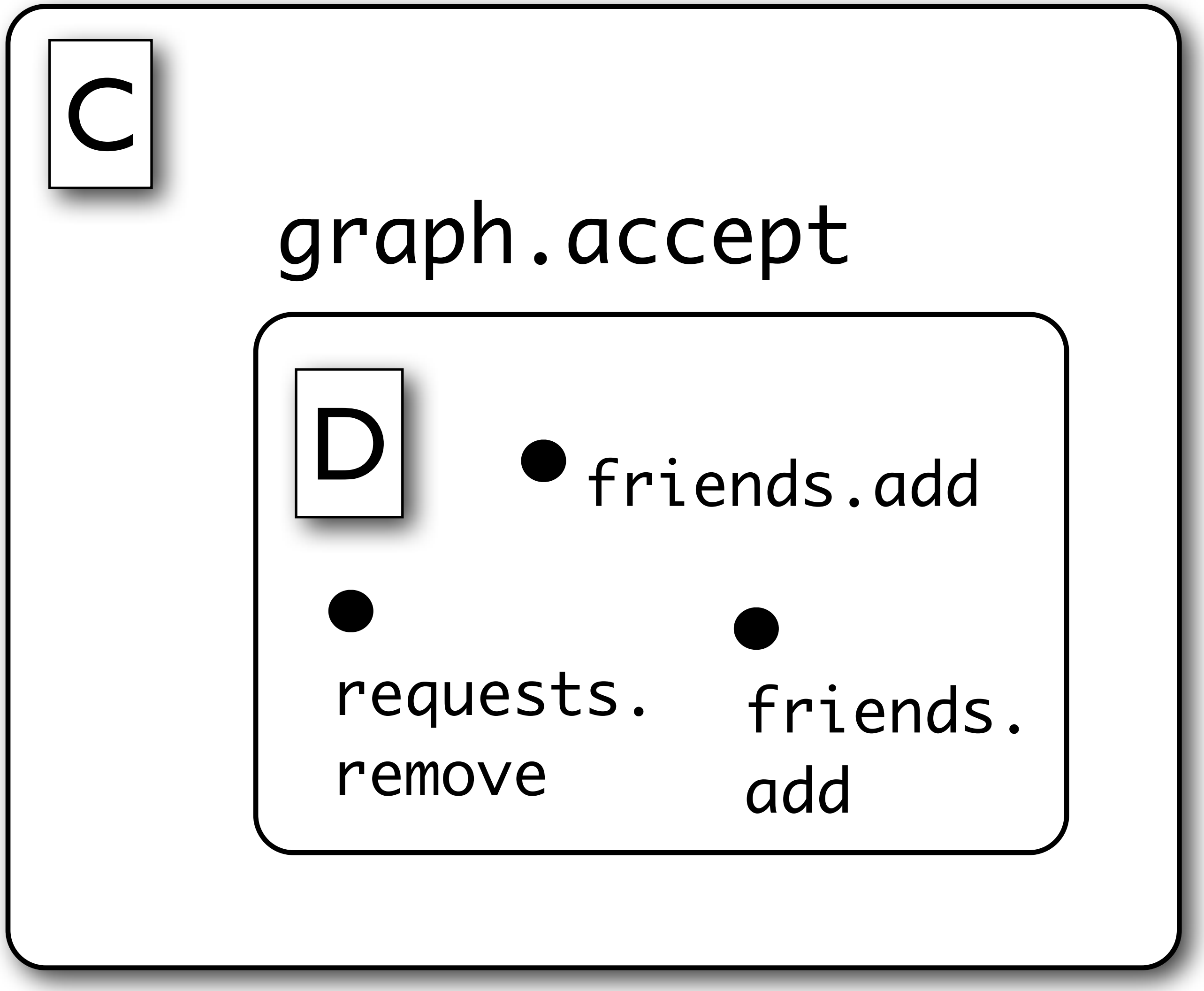
```
primitive datatype SocialGraph

operation accept(Bob → Alice)
```

Existing spec mechanism
[POPL'14]

Object of a composite data type D ➜ Object of a primitive data type with a spec ⟦D⟧

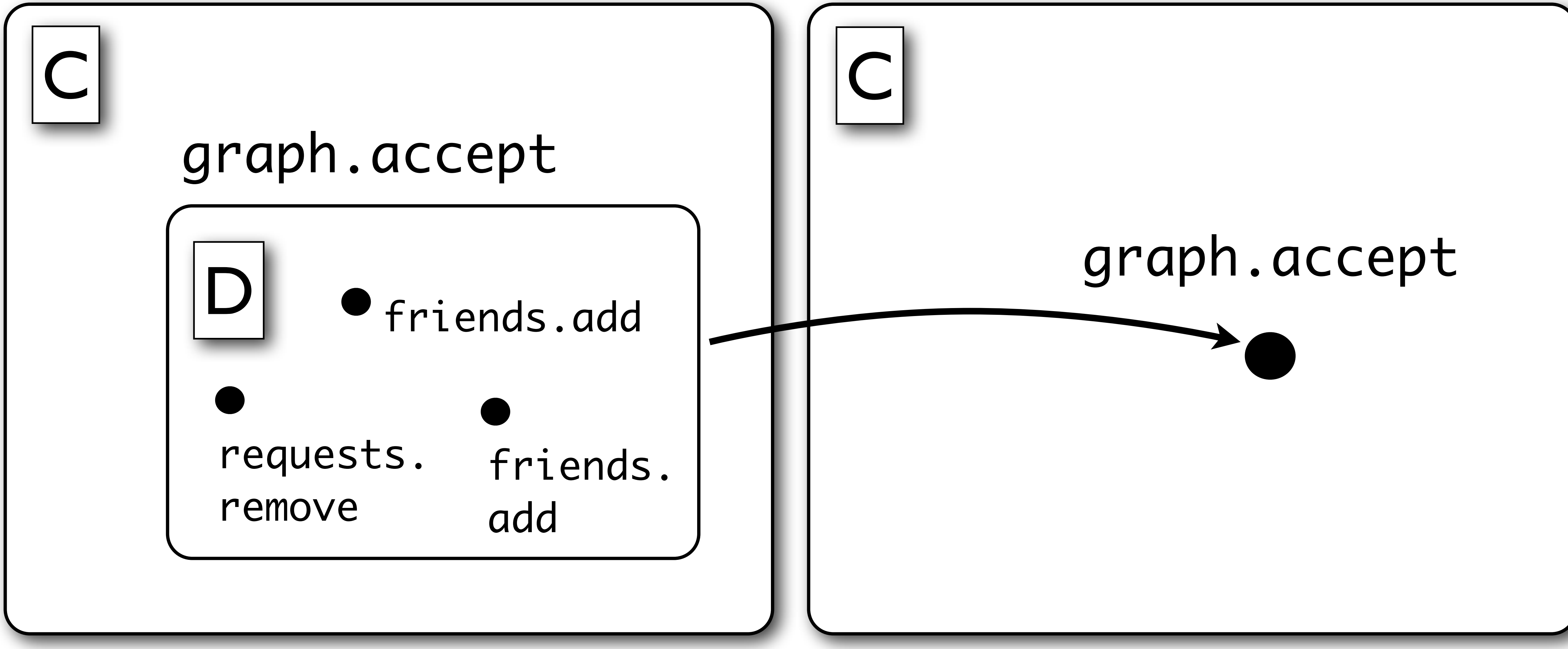
Denotation ⟦D⟧ of D
= best spec



Fine-grain semantics

⟦C(D)⟧

Method invocation ➜
multiple events
on internal objects

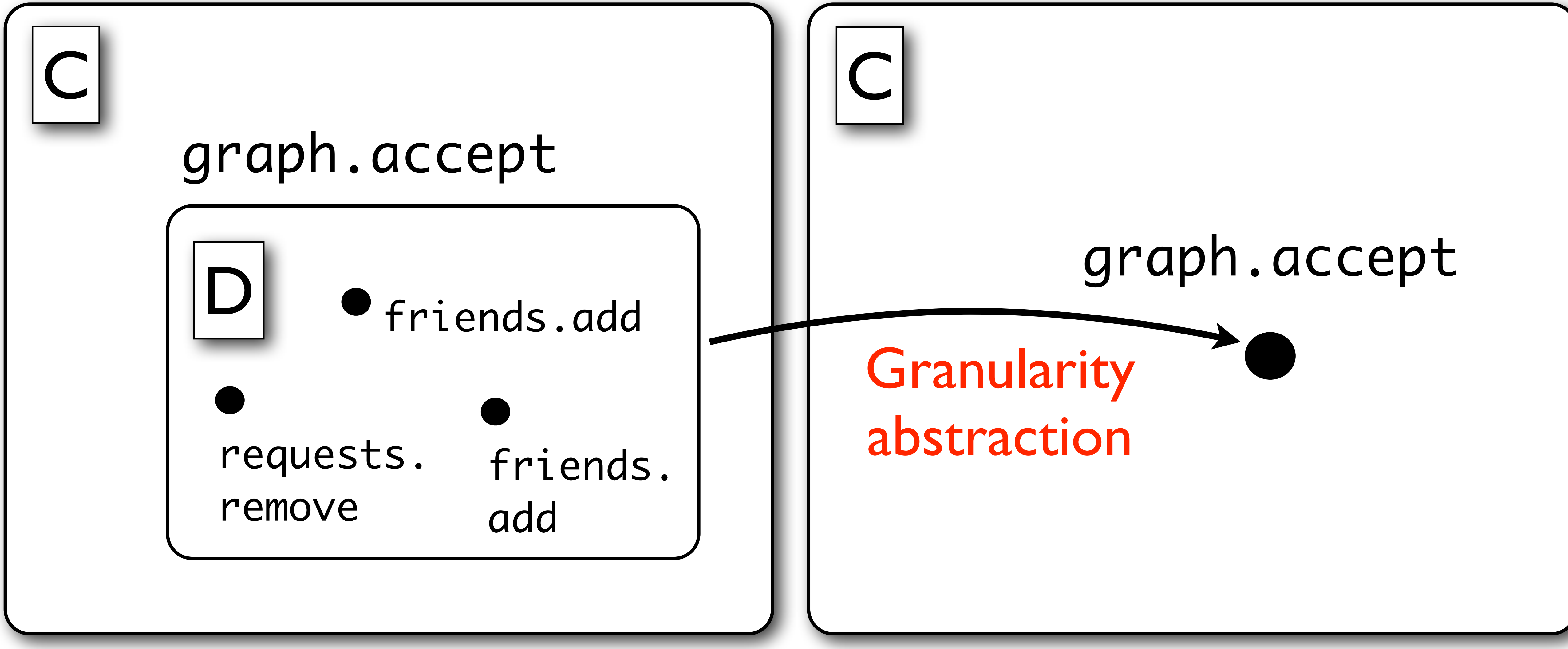


Fine-grain semantics
⟦C(D)⟧

Coarse-grain denotational semantics ⟪C(⟦D⟧)⟫

Method invocation ➜
multiple events
on internal objects

Method invocation ➜
one event

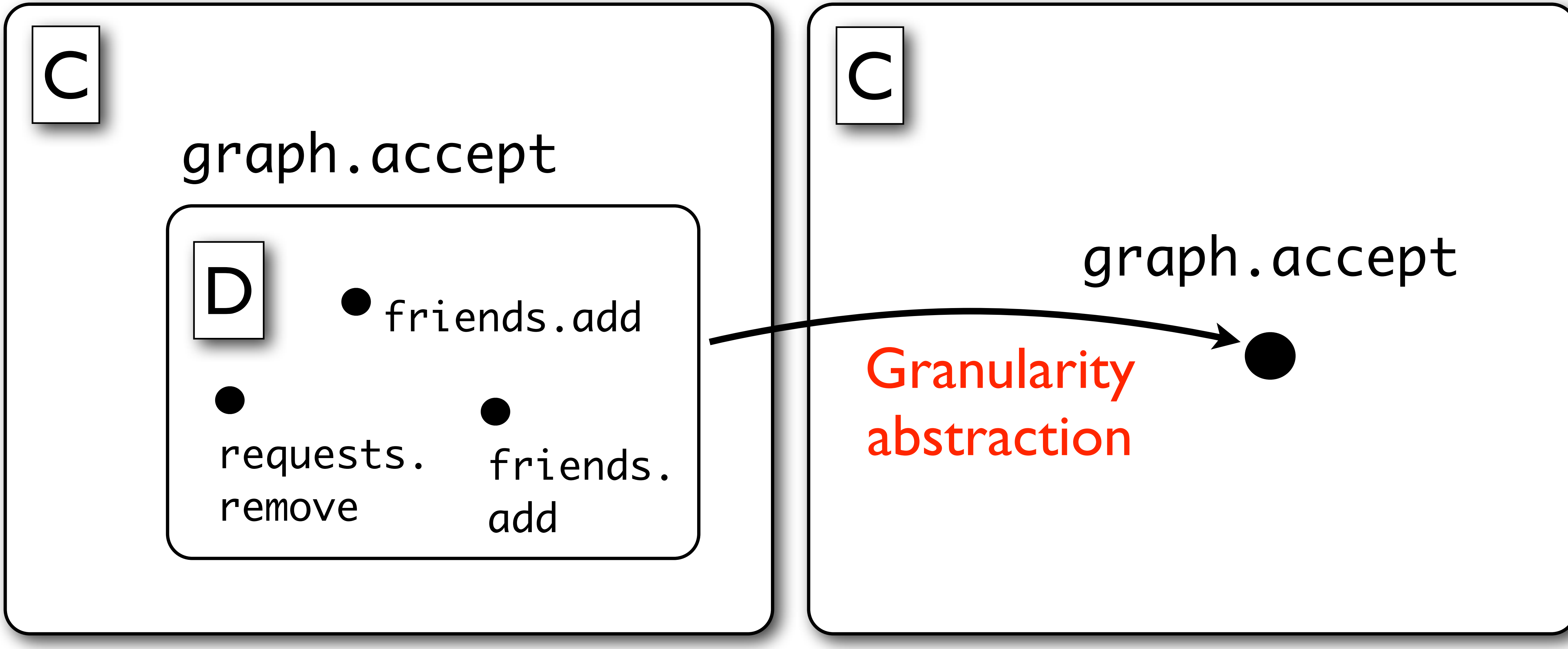


Fine-grain semantics
⟦C(D)⟧

Coarse-grain denotational semantics 《C(⟦D⟧)》

Method invocation ➜ multiple events on internal objects

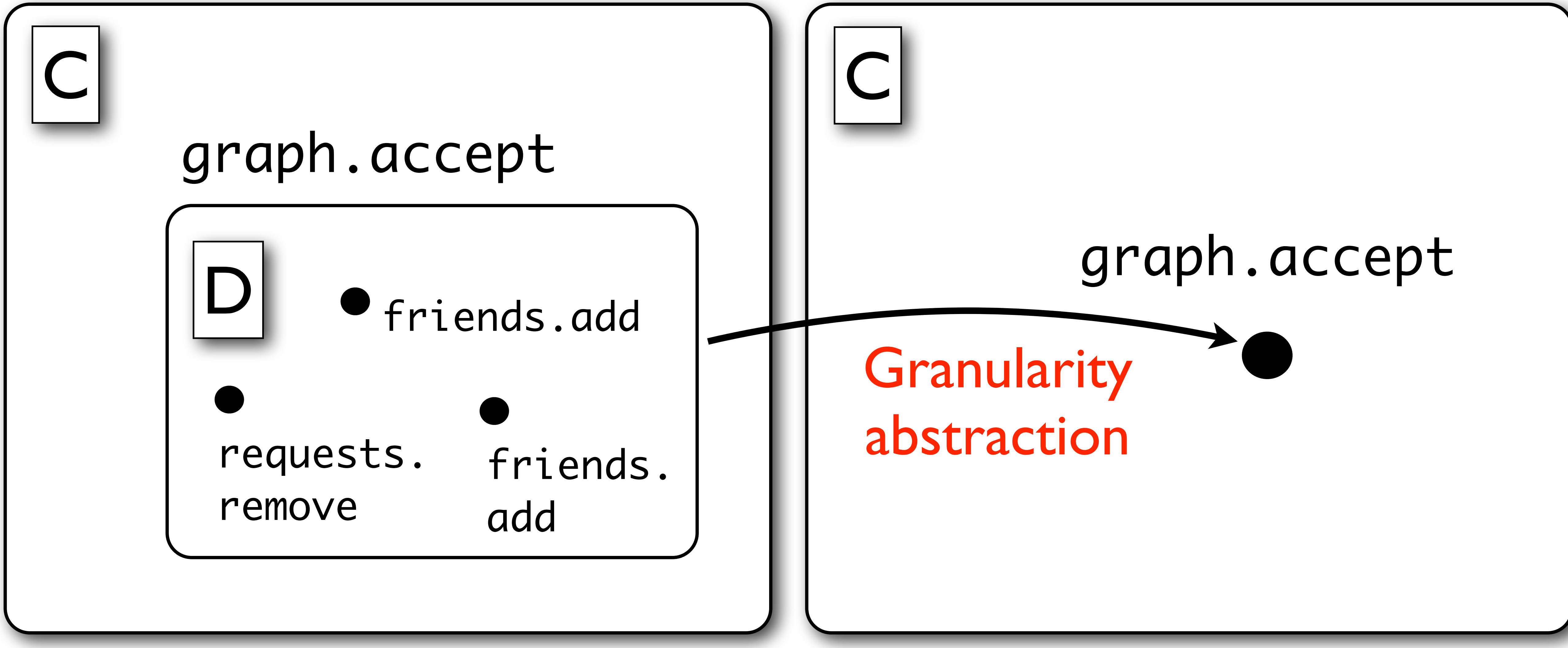
Method invocation ➜ one event



Fine-grain semantics
⟦C(D)⟧

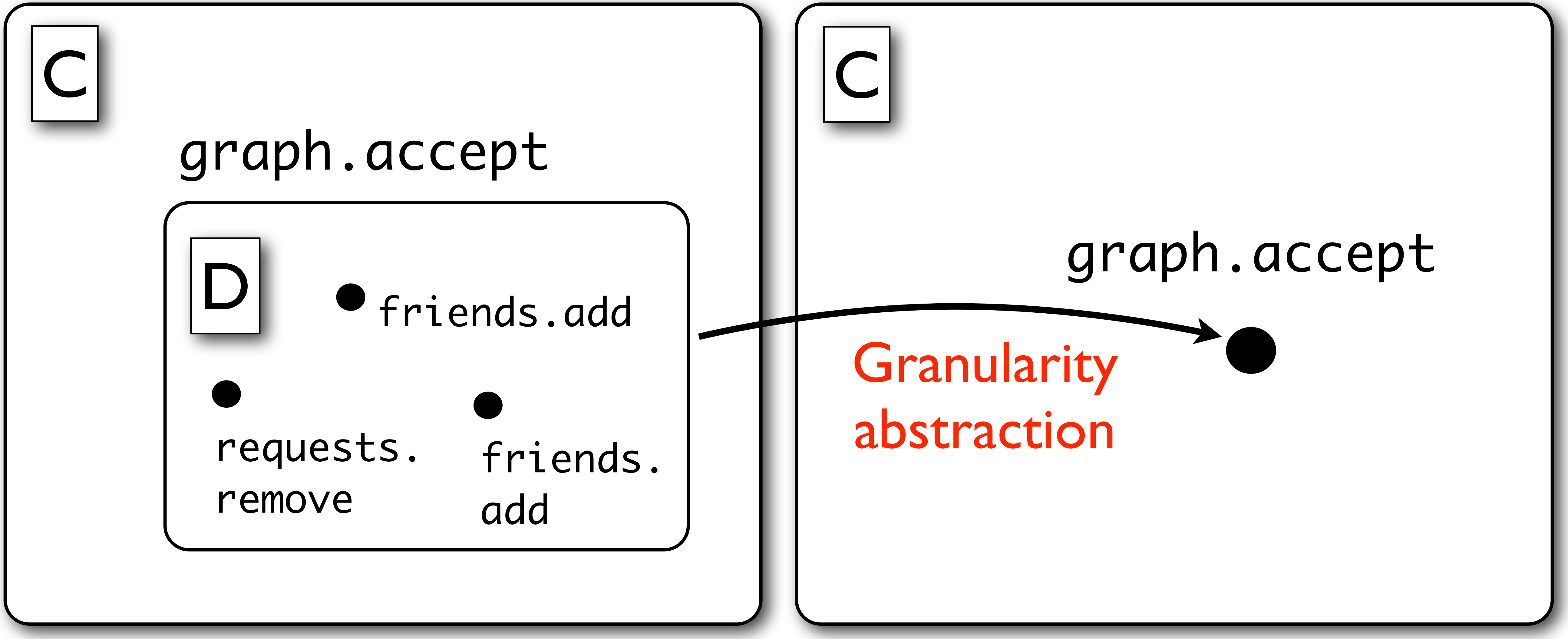
Coarse-grain denotational
semantics 《C(⟦D⟧)》

Method results determined using the spec ⟦D⟧
instead of the implementation D



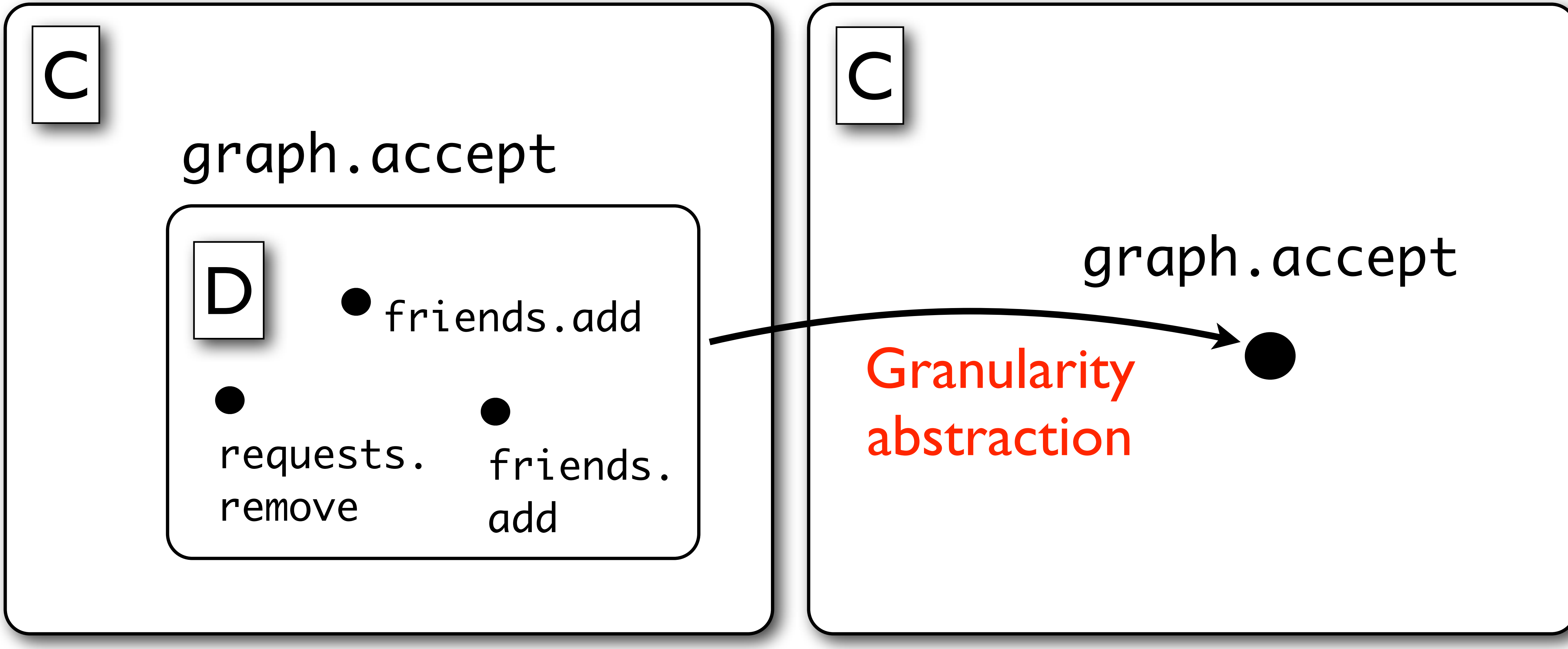
Fine-grain semantics ⟦C(D)⟧ = Coarse-grain denotational semantics 《C(⟦D⟧)》

Method results determined using the spec ⟦D⟧ instead of the implementation D



Fine-grain semantics ⟦C(D)⟧ = Coarse-grain denotational semantics 《C(⟦D⟧)》

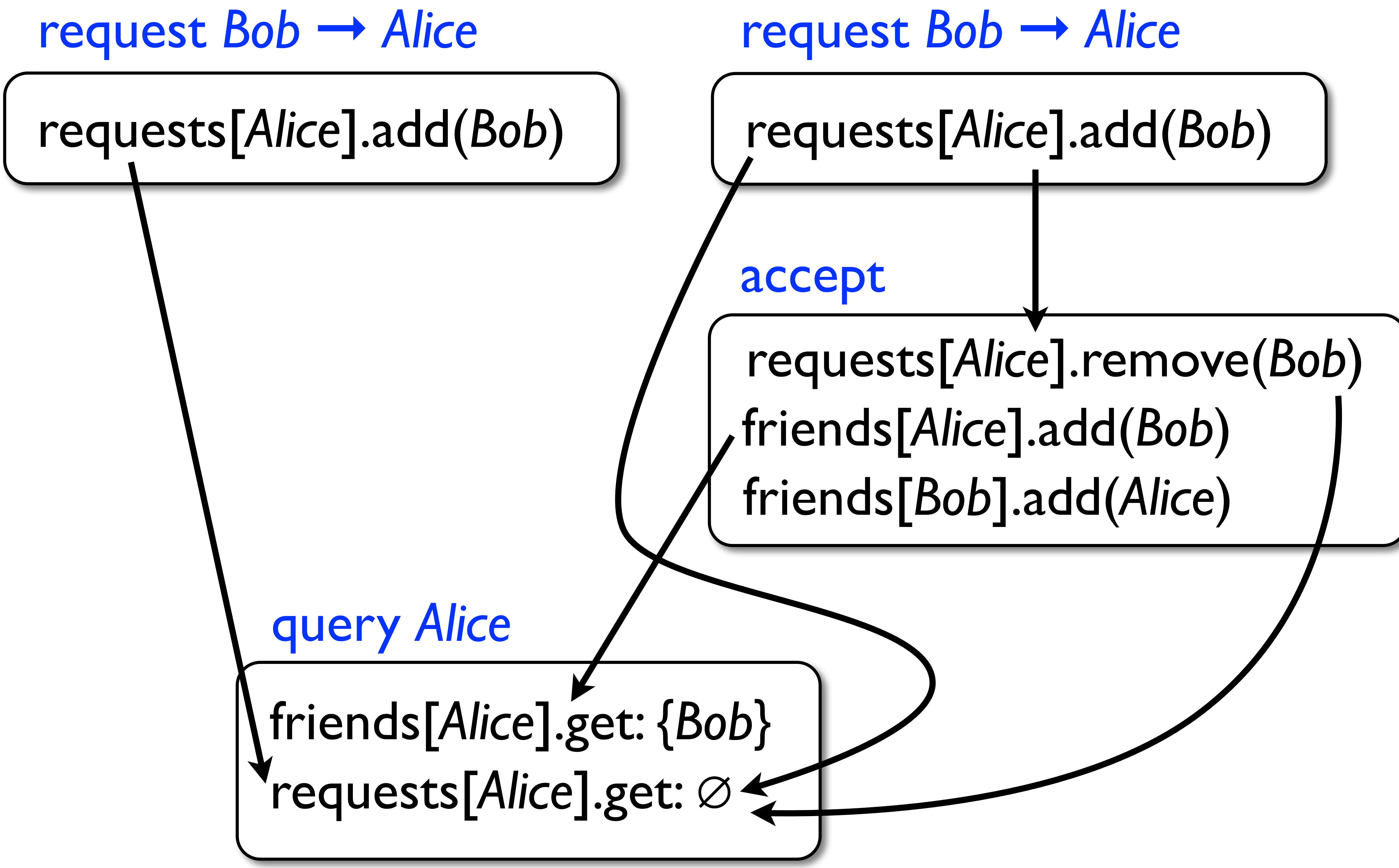
1. Semantics of the bare database with primitive replicated data types



Fine-grain semantics ⟦C(D)⟧ = Coarse-grain denotational semantics 《C(⟦D⟧)》

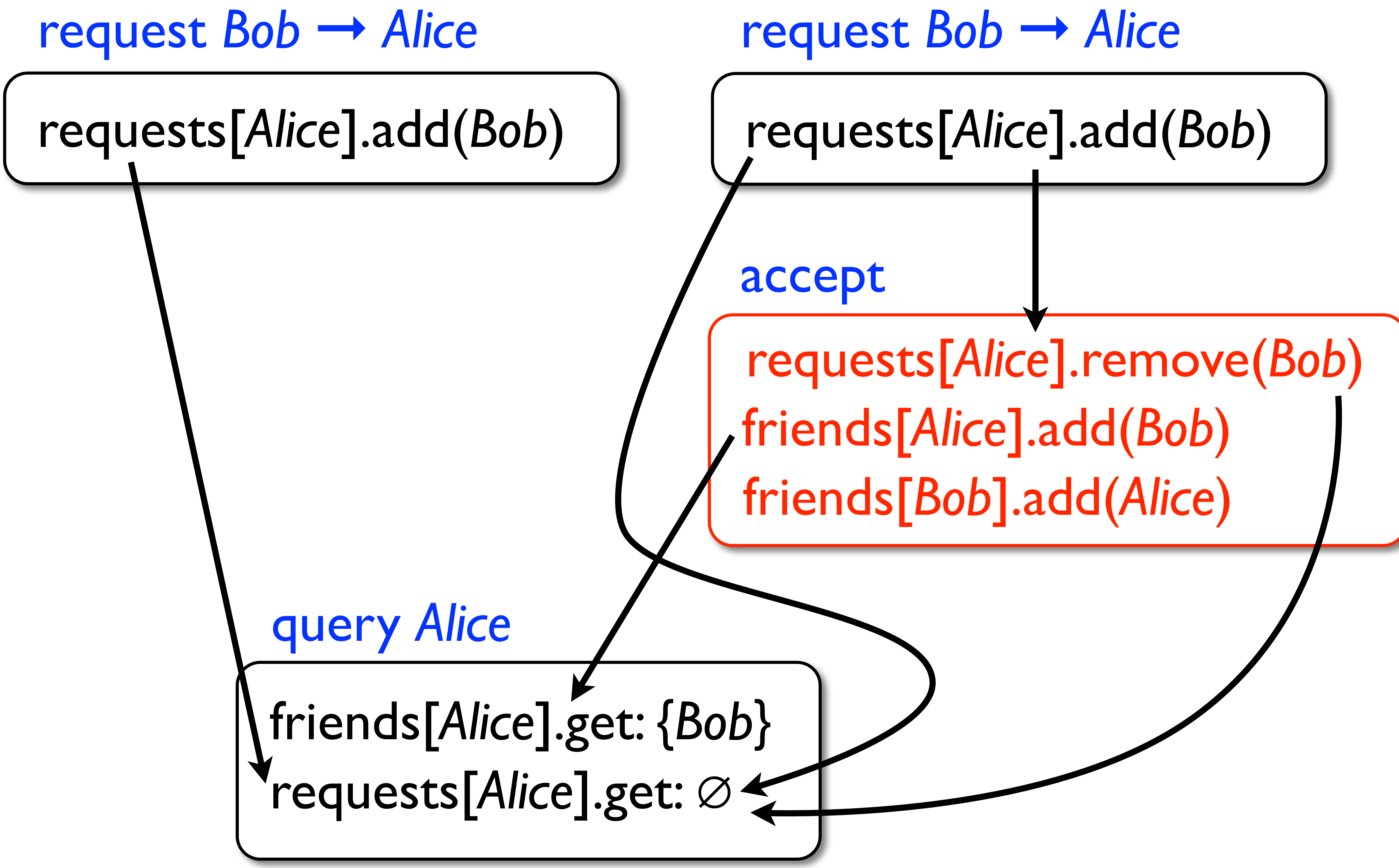
1. Semantics of the bare database with primitive replicated data types

2. Definition of ⟦D⟧

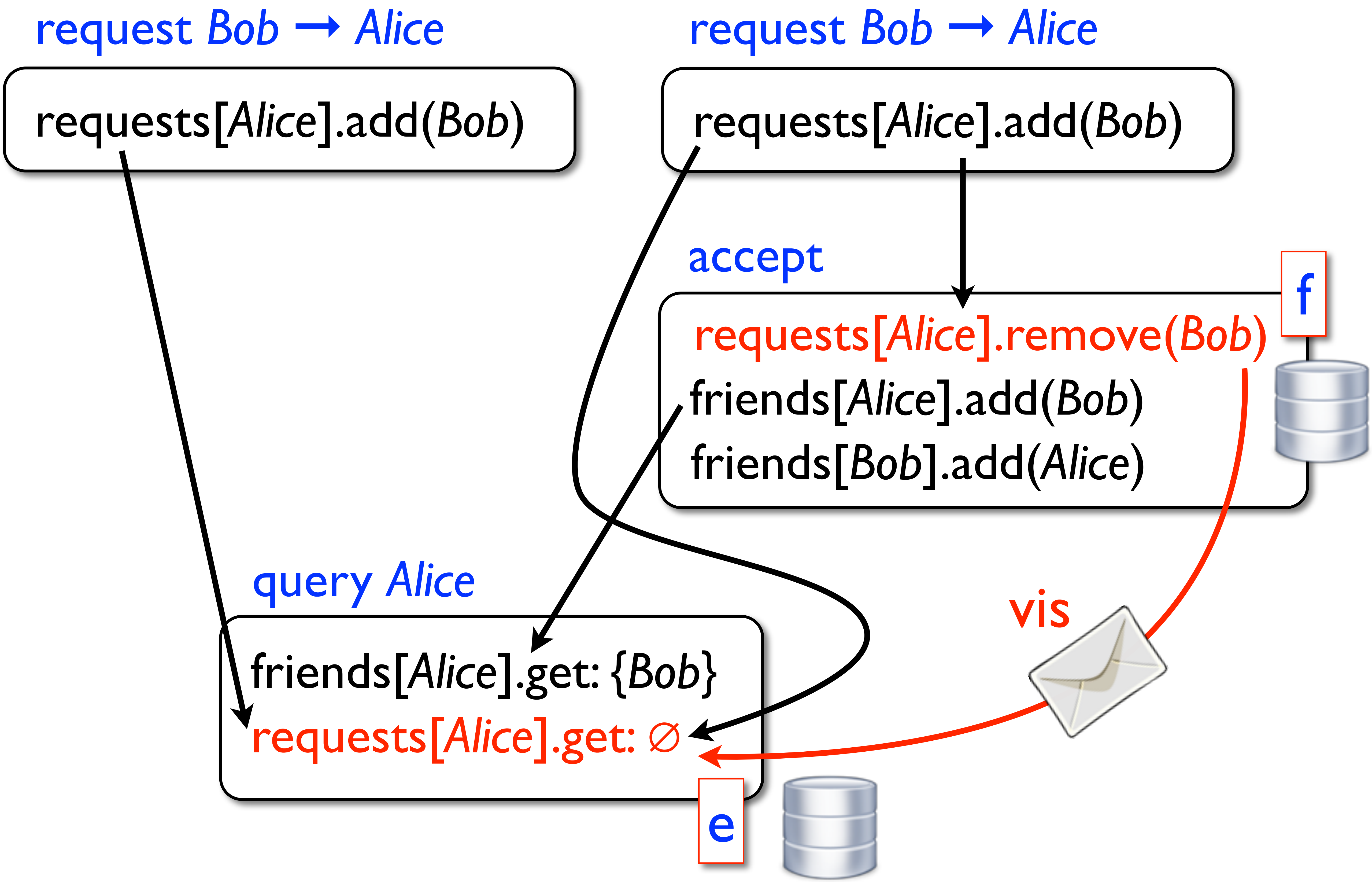


Database semantics = set of executions ~ graphs

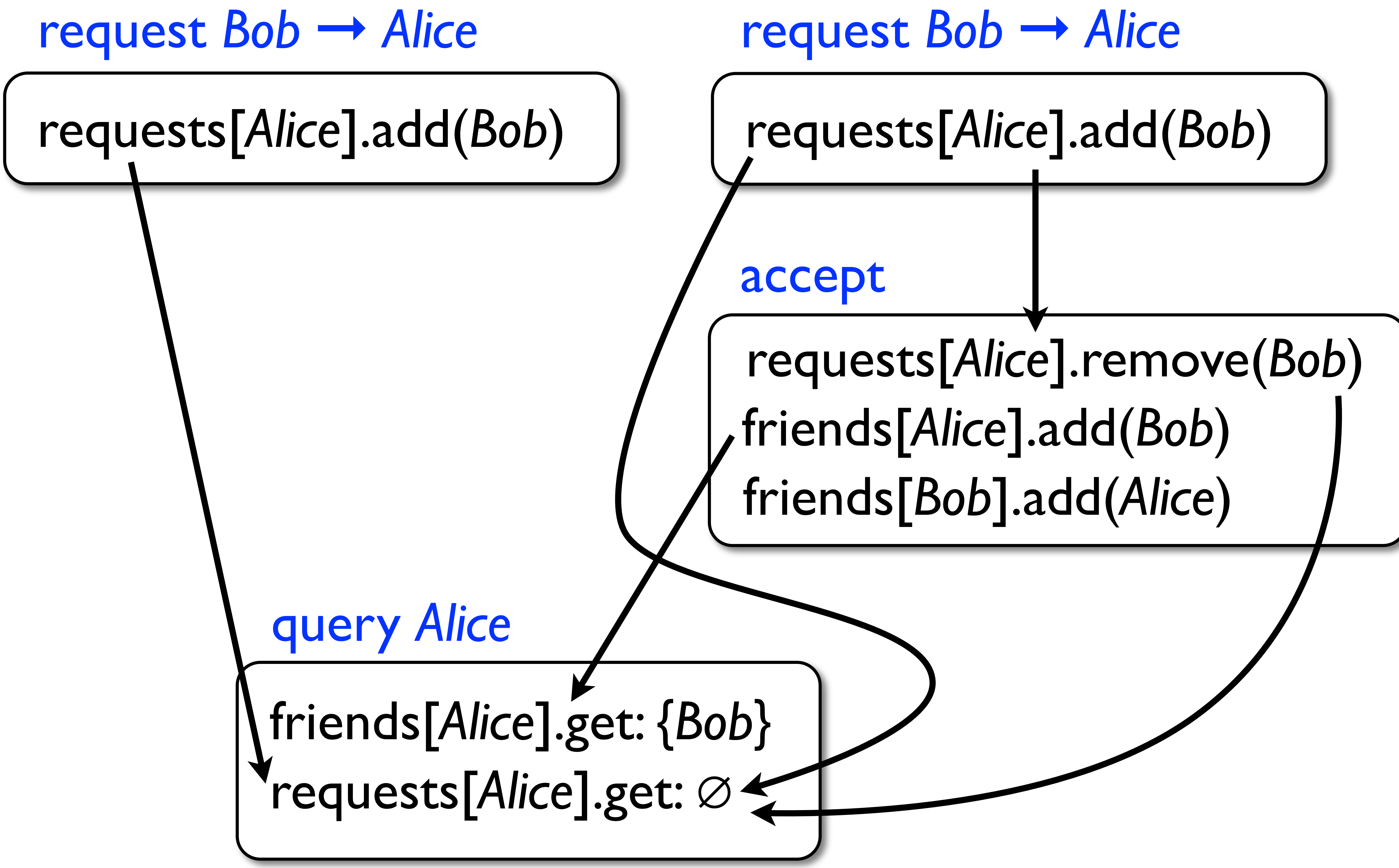
Nodes: events describing operations performed



Events grouped into transactions

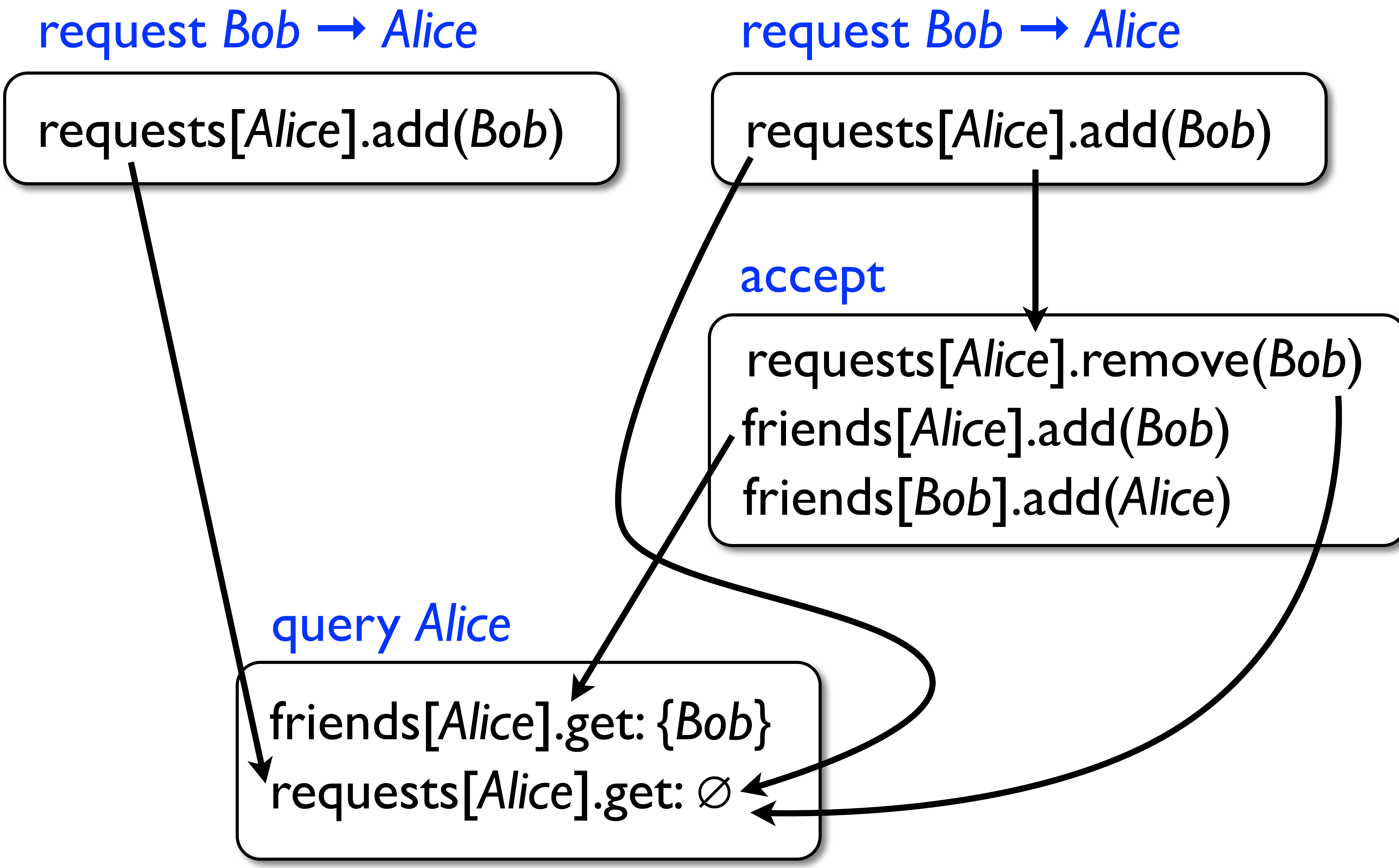


Visibility: (f, e) ∈ vis ➜ f delivered to the replica of e



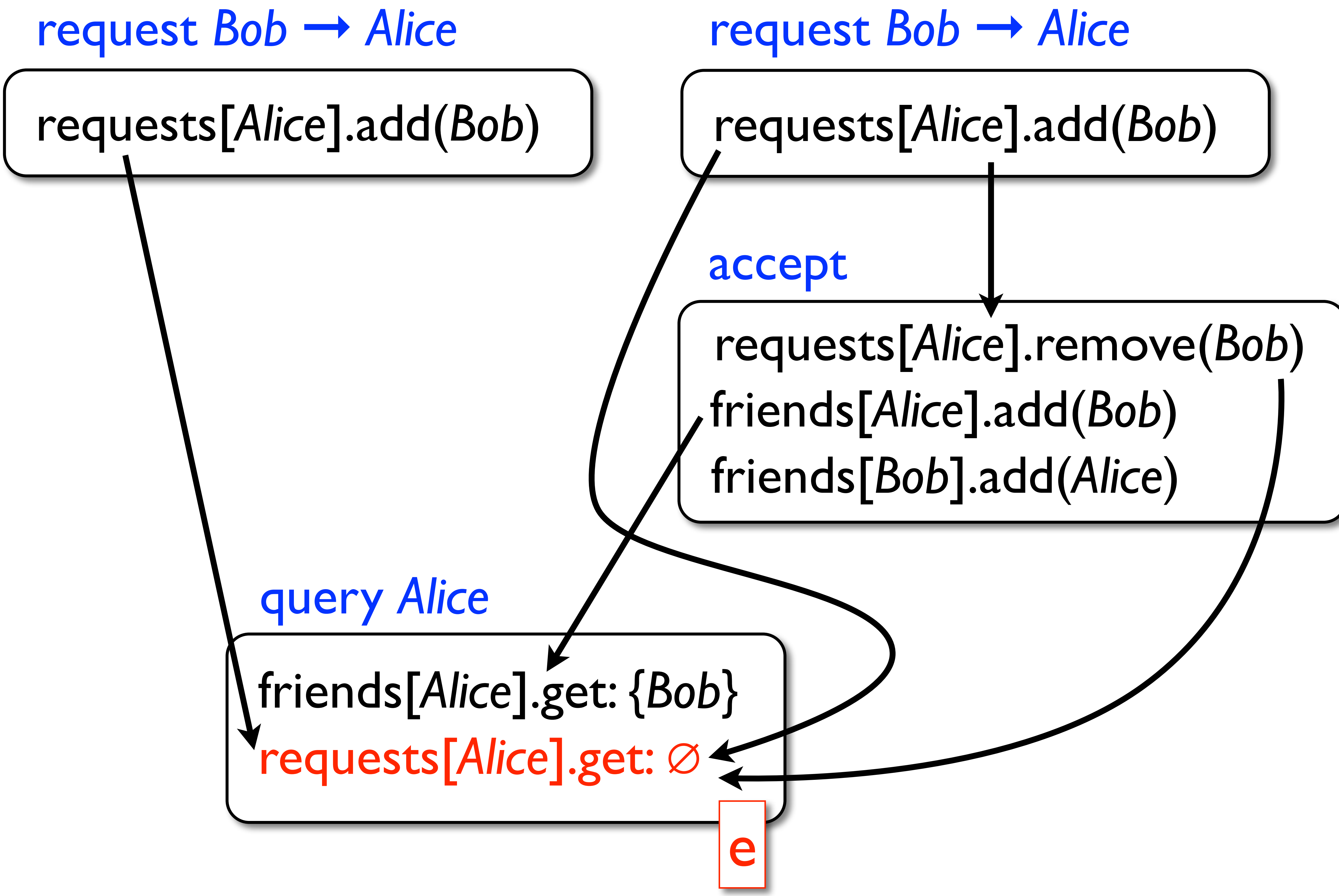
Visibility satisfies consistency axioms: e.g., transitivity

Restrict the anomalies allowed



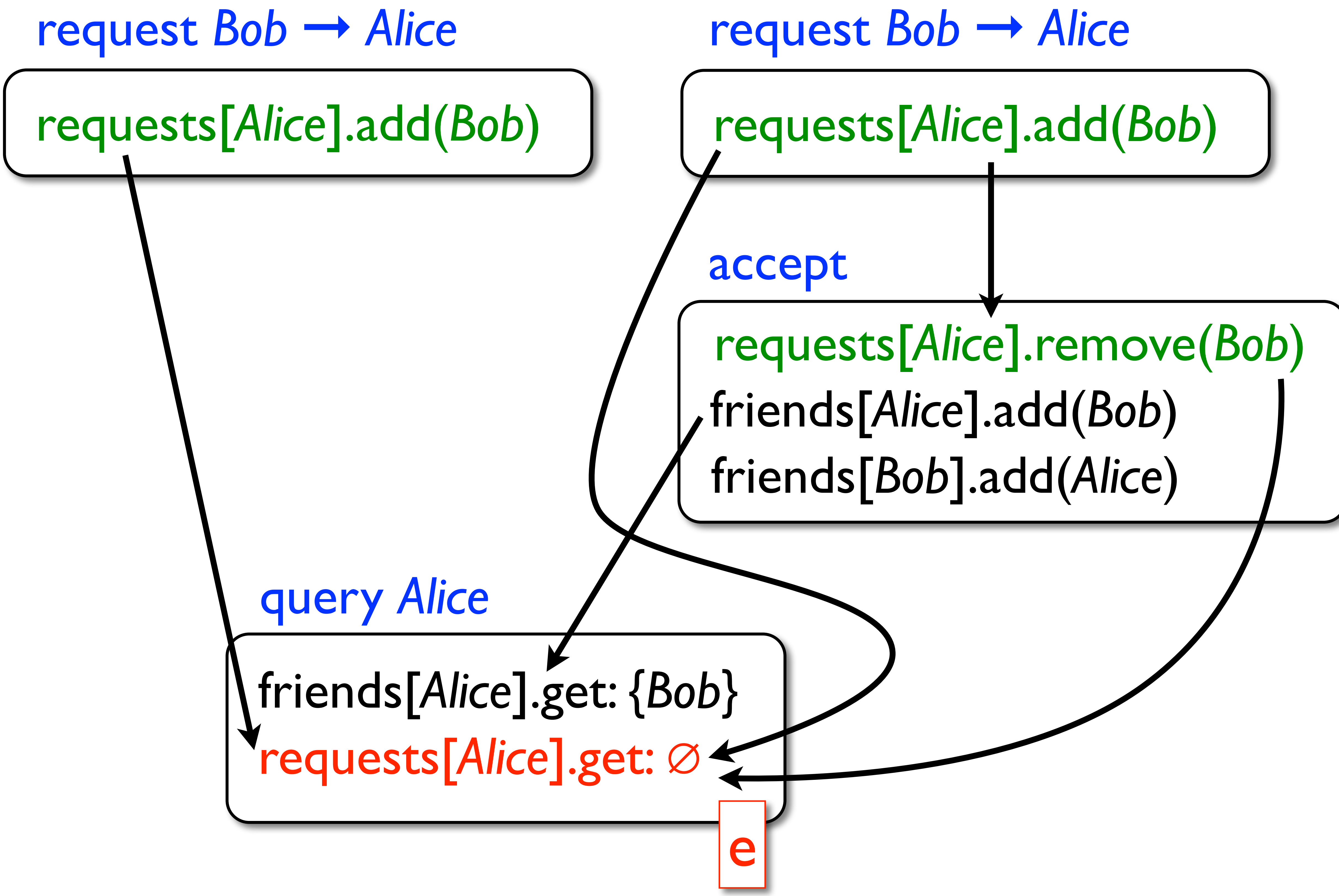
Database semantics = executions with correct return values

Determined by replicated data type specifications



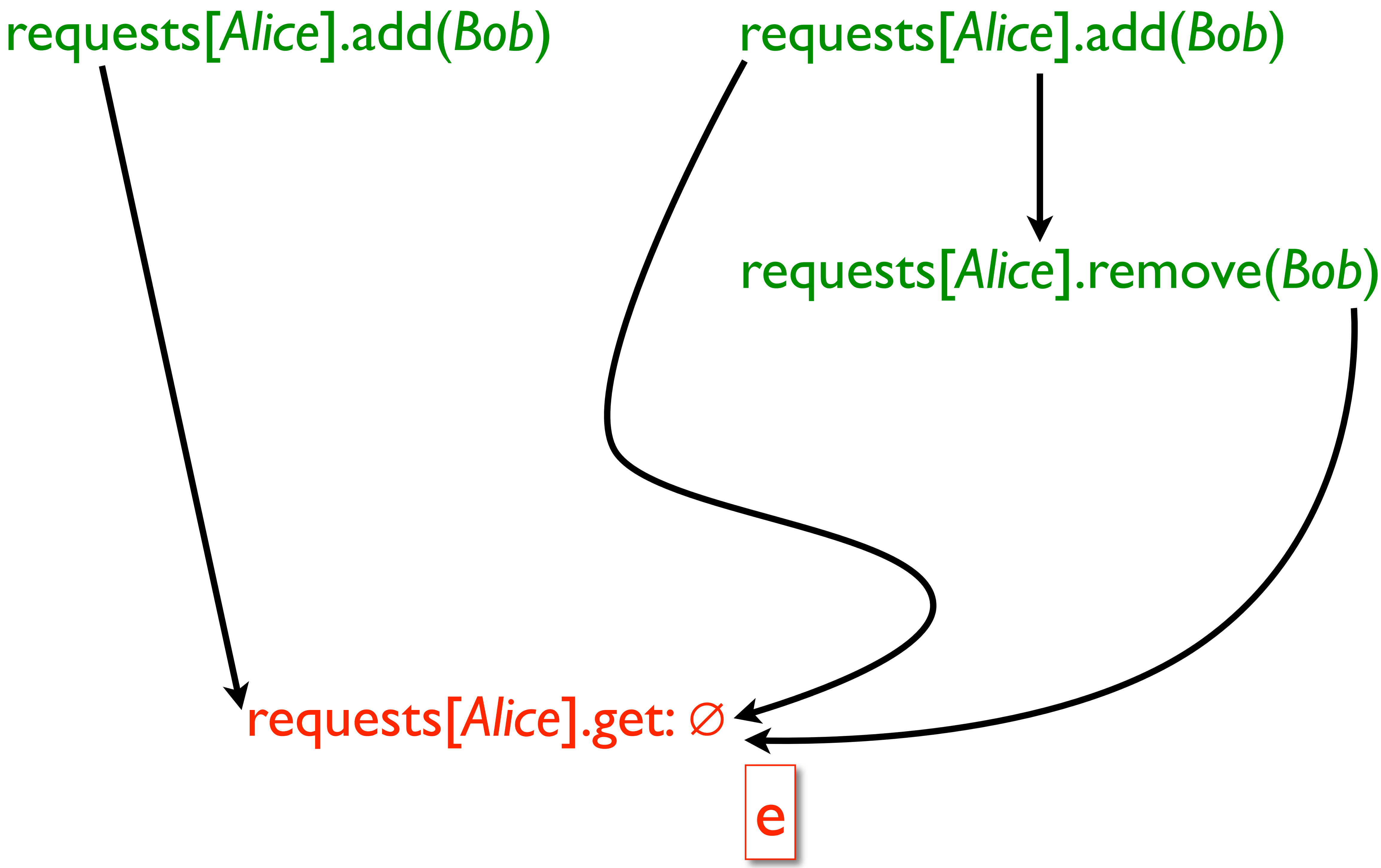
Database semantics = executions with correct return values

Determined by replicated data type specifications



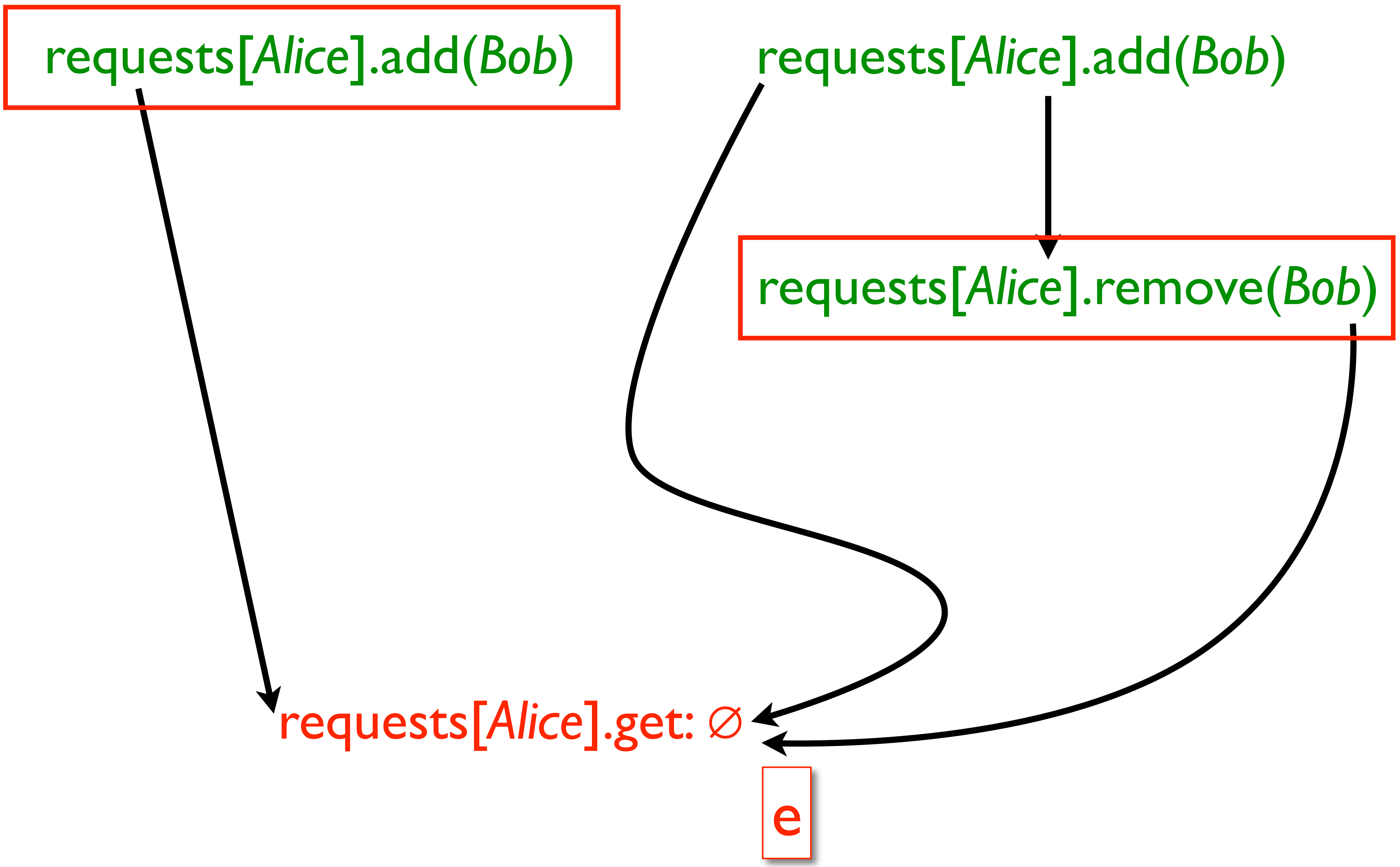
Database semantics = executions with correct return values

Determined by replicated data type specifications



Context(e): projection to events visible to e

Spec S: Context(e) → return value(e)

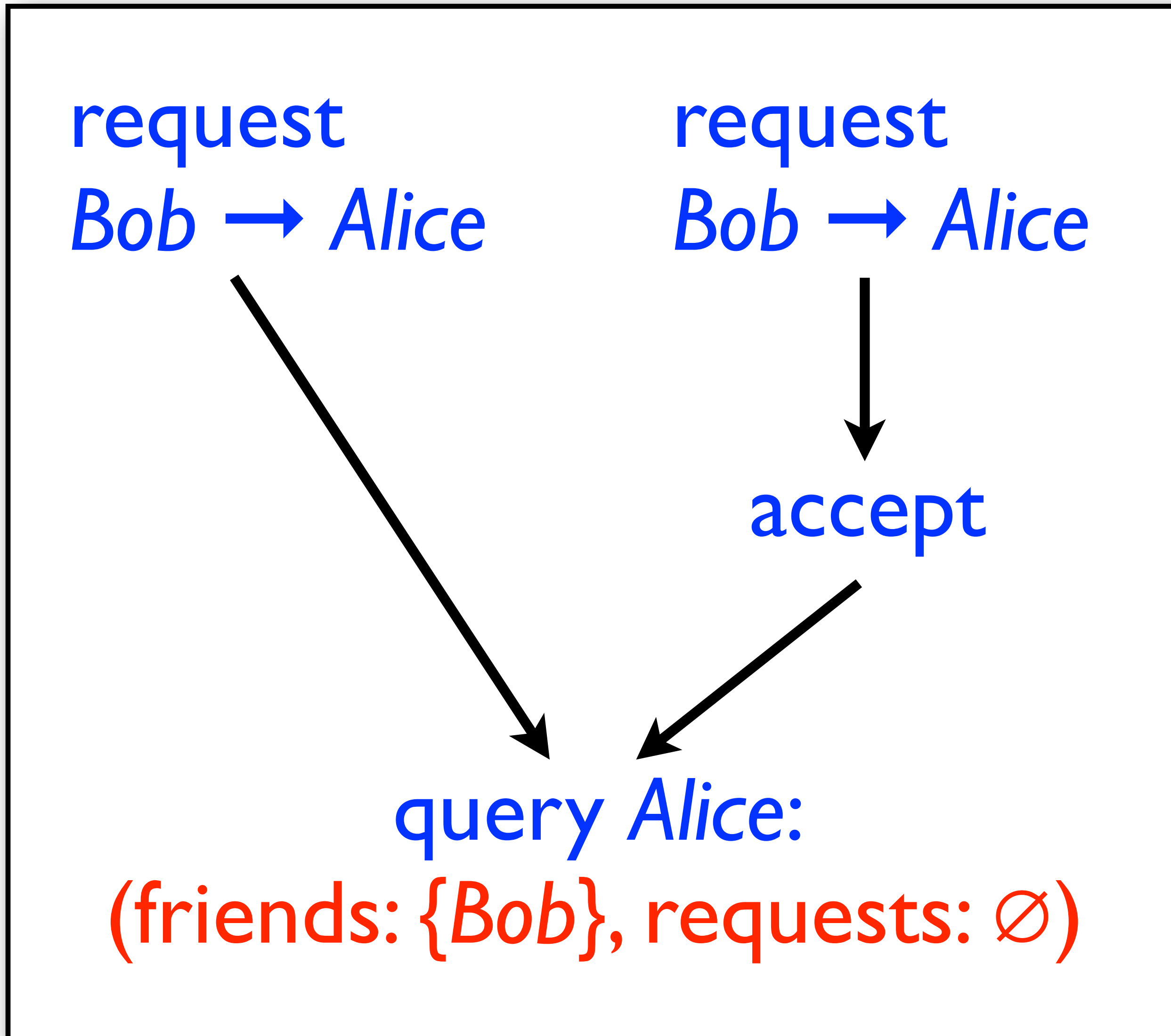


Remove-wins set: removes cancel adds unrelated to them in visibility

Contribution: D ➜ ⟦D⟧ = S ∈ Context → Value

```
datatype SocialGraph {
 RemoveWinsSet friends[];
 RemoveWinsSet requests[];

 tx accept(Bob→Alice) {
  ...
  requests[Alice].remove(Bob);
  friends[Alice].add(Bob);
  friends[Bob].add(Alice);
  ...
 }
}
```

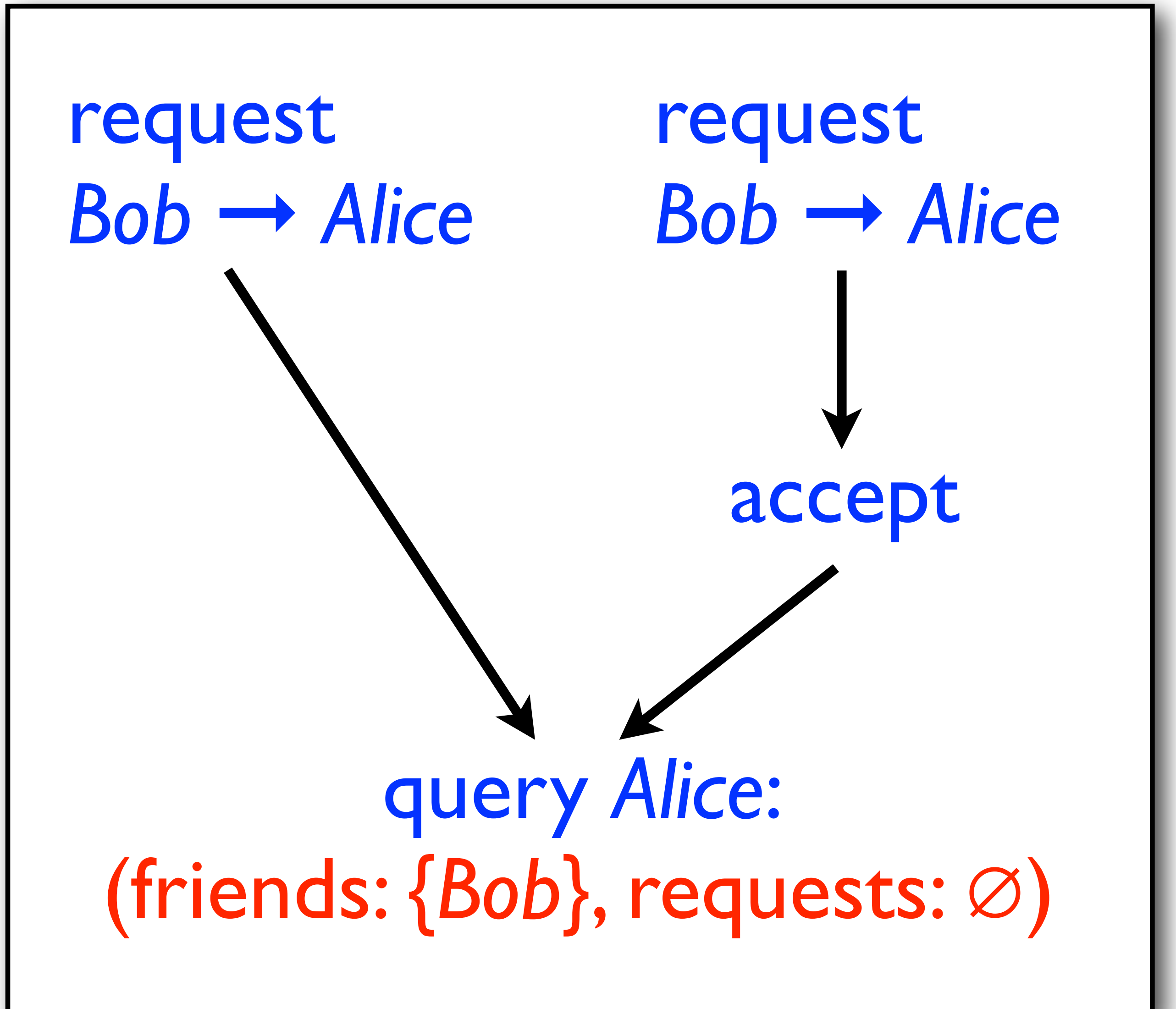


- Compute method return value based on coarse-grain events ~ executed methods
- No information about fine-grain internal events

Contribution: $D \rightarrow [\![D]\!] = S \in \text{Context} \rightarrow \text{Value}$

```
datatype SocialGraph {
 RemoveWinsSet friends[];
 RemoveWinsSet requests[];

 tx accept(Bob→Alice) {
  ...
  requests[Alice].remove(Bob);
  friends[Alice].add(Bob);
  friends[Bob].add(Alice);
  ...
 }
}
```



- Compute method return value based on coarse-grain events ~ executed methods
- No information about fine-grain internal events

Coarse-grain context N

request *Bob* → *Alice*

↓

request *Bob* → *Alice*

↓

accept

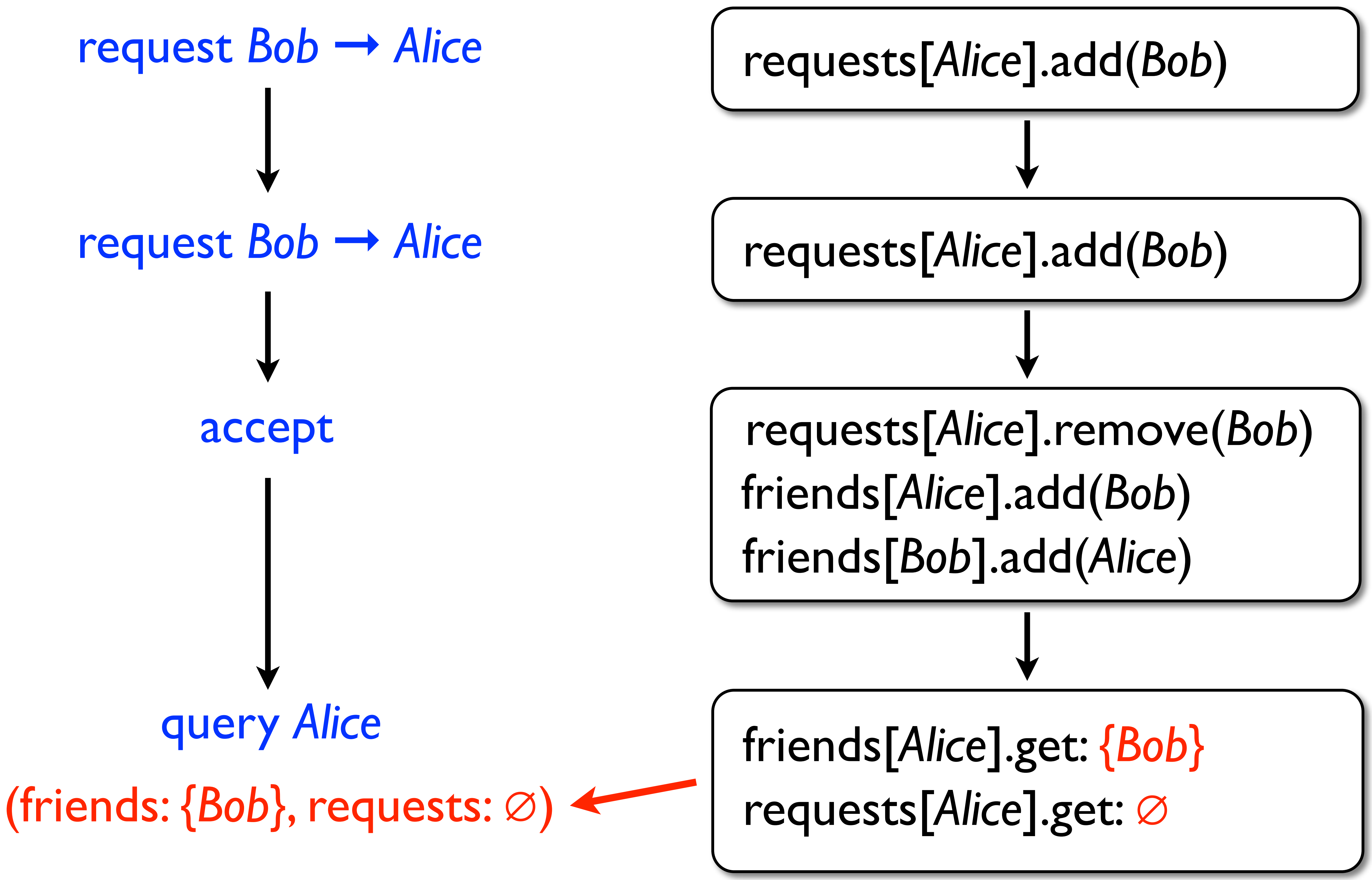
↓

query *Alice*

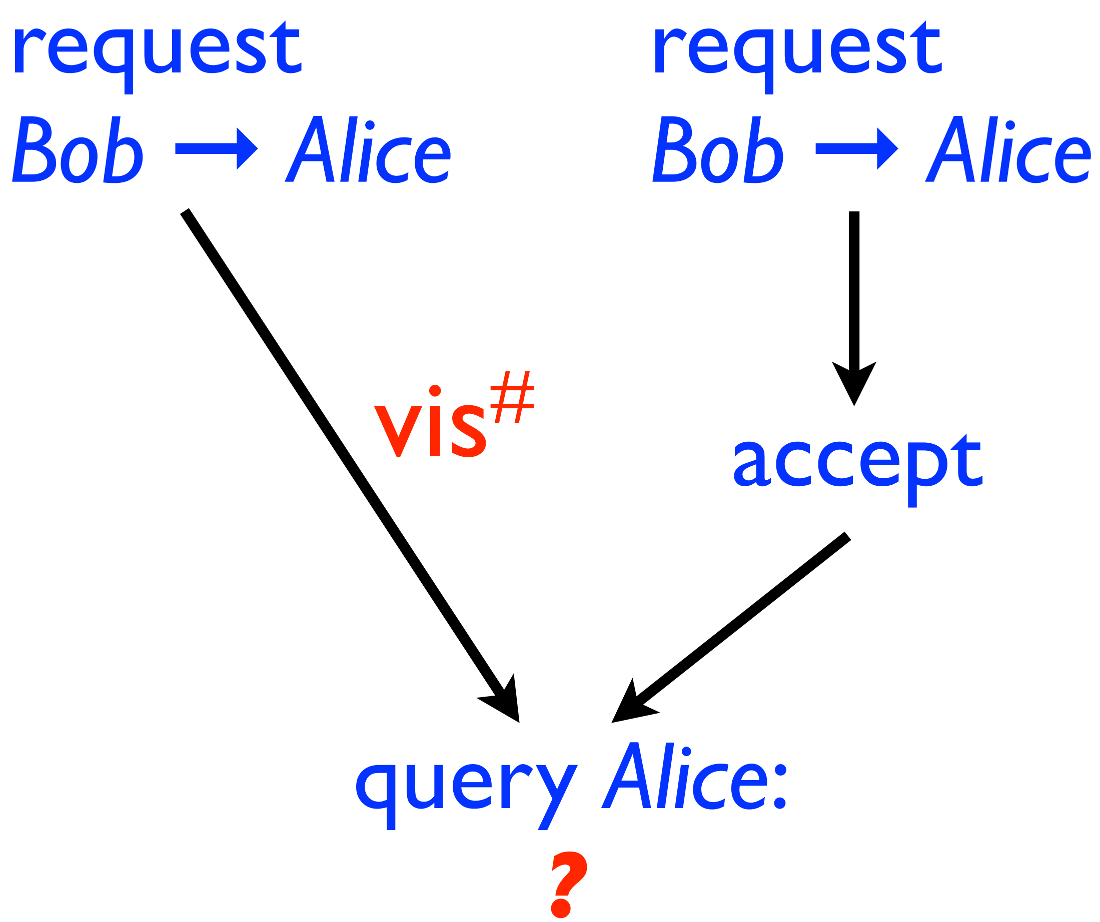
?

Coarse-grain context N ➜ Fine-grain execution ɣ(N)

Coarse-grain context N ➜ Fine-grain execution ɣ(N)

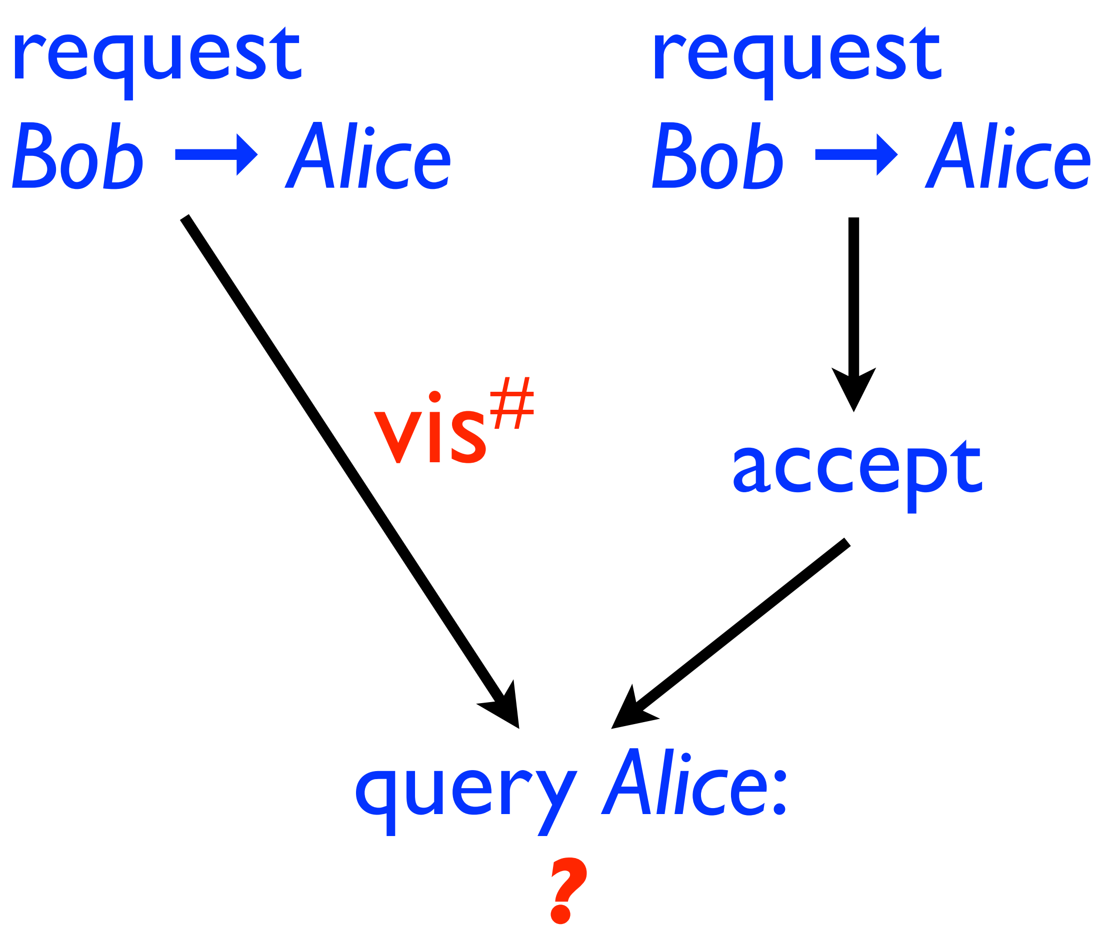


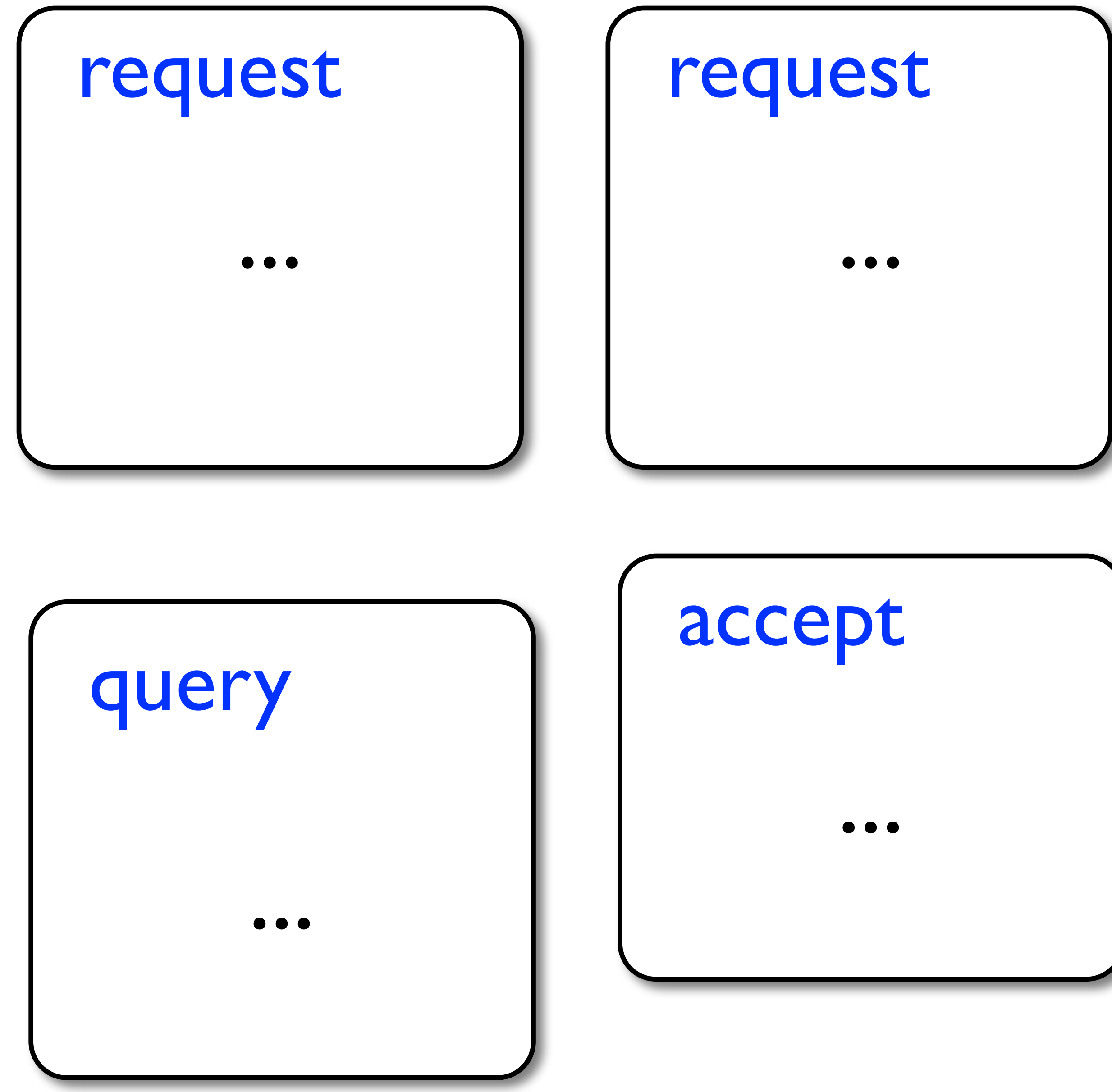
Coarse-grain context N



Given visibility vis# on whole methods = txns

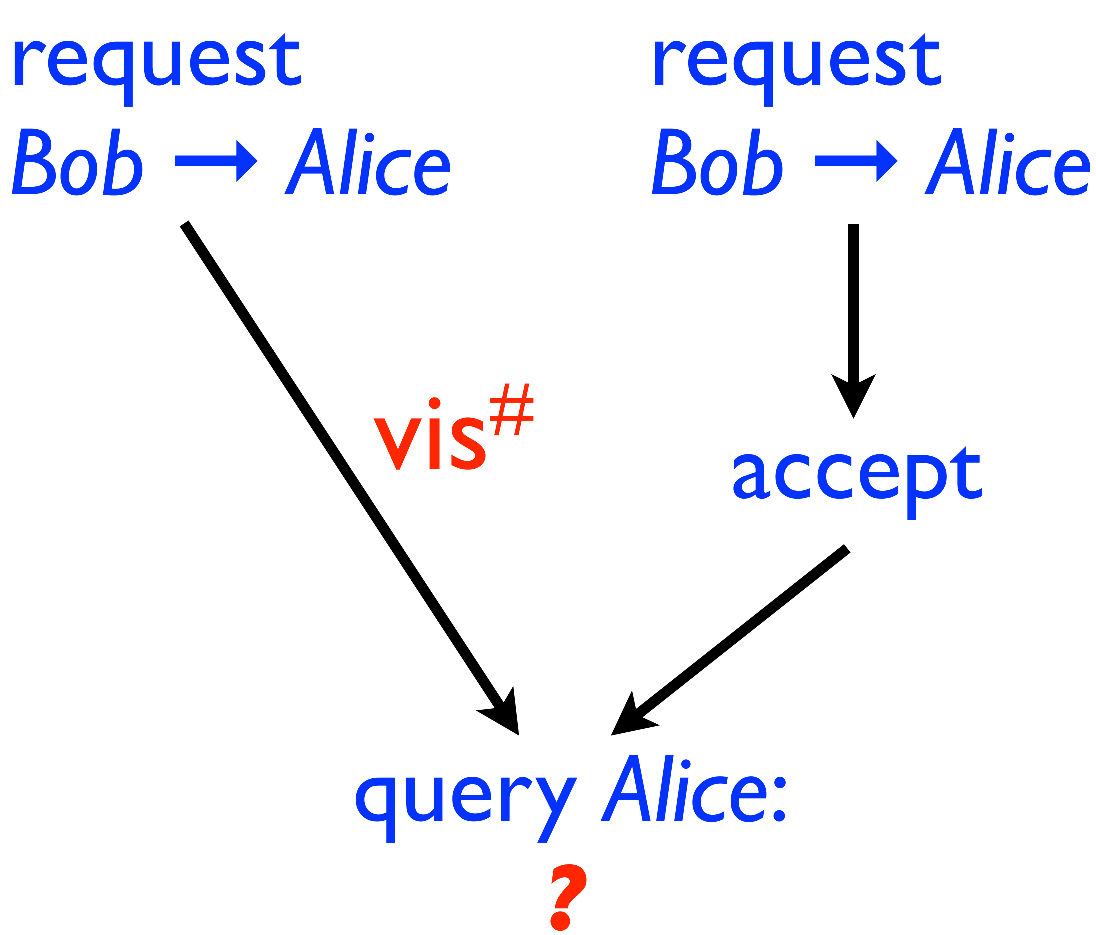
Coarse-grain context N ➜ Fine-grain execution $\gamma(N)$

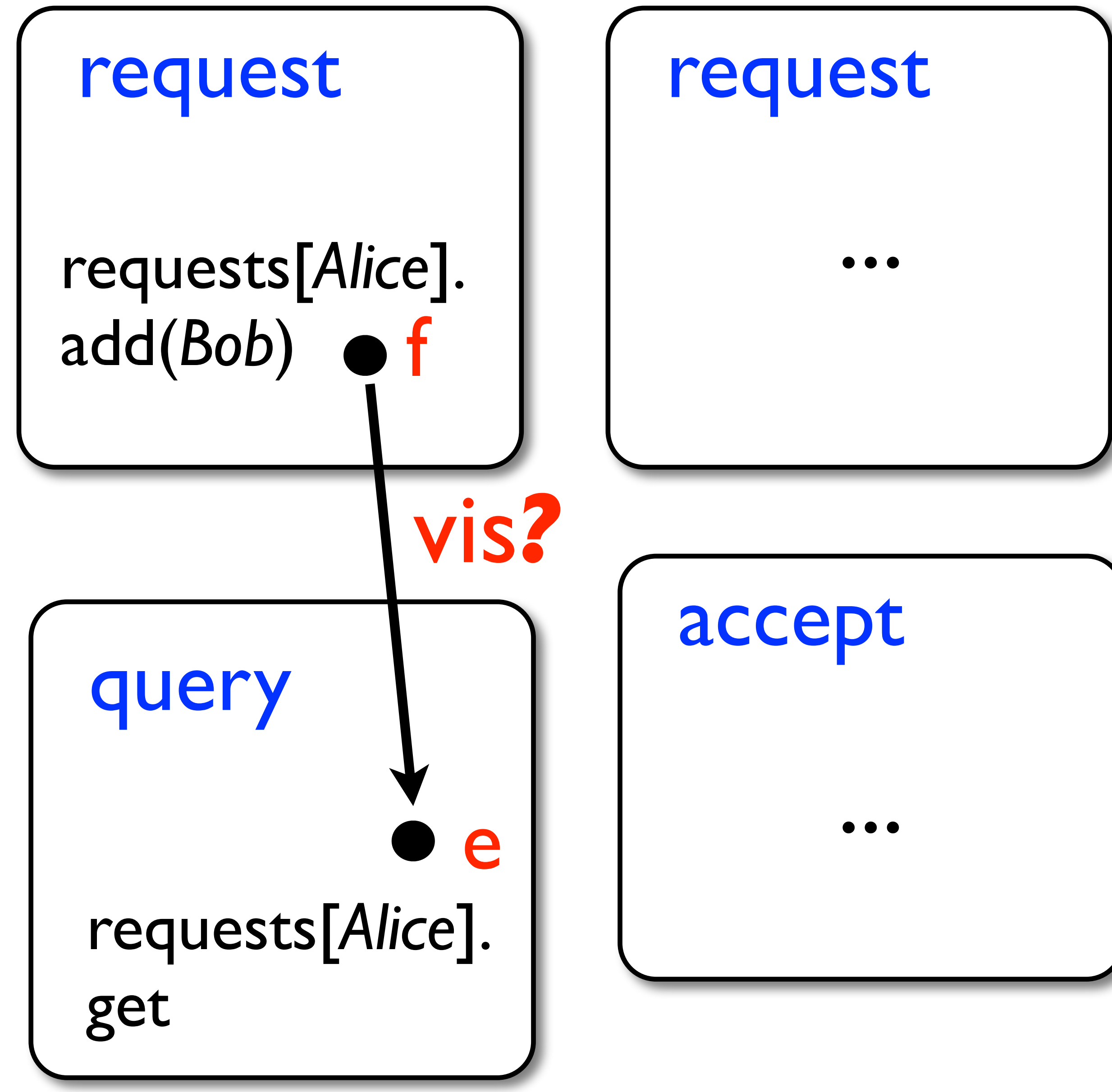




Given visibility $vis^{\#}$ on whole methods = txns

Coarse-grain context N ➜ Fine-grain execution ɣ(N)

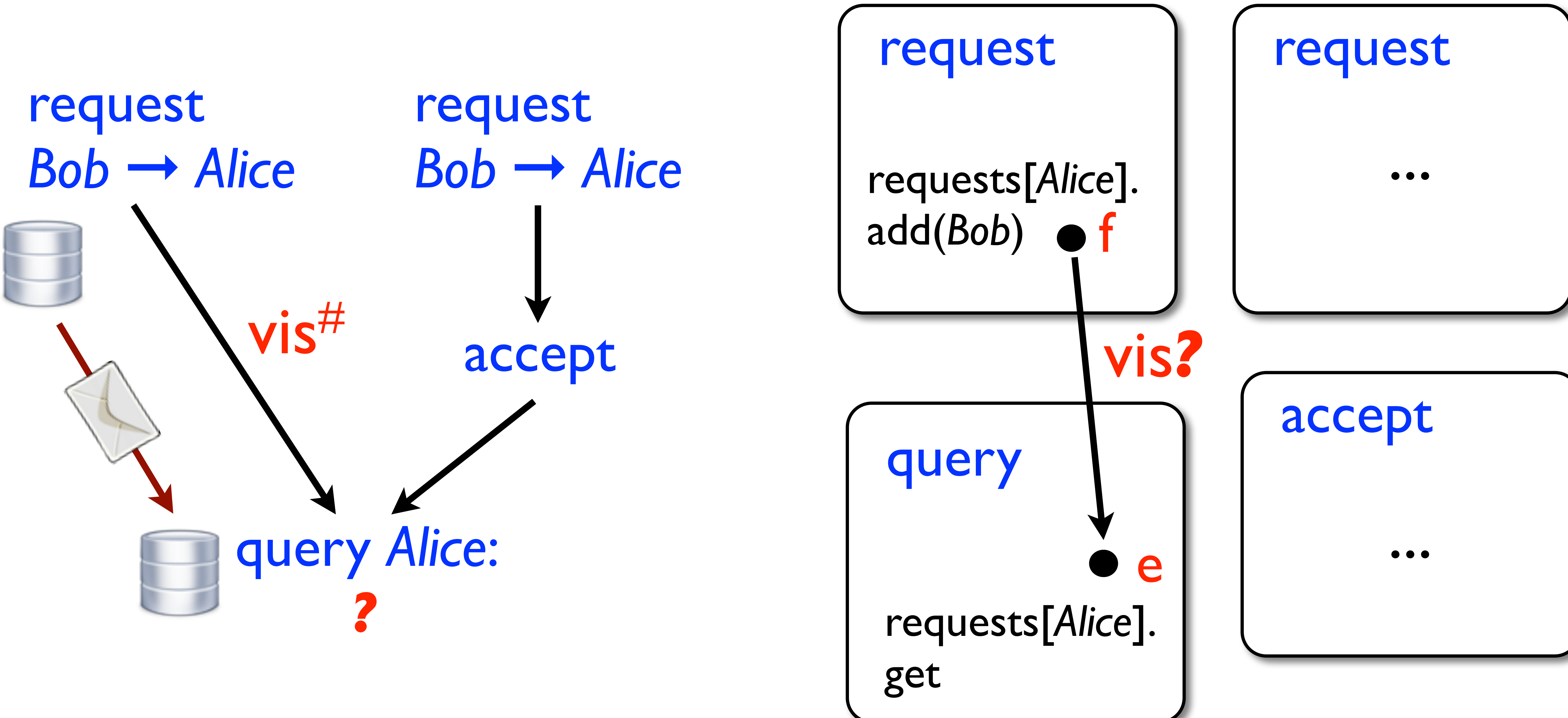




Given visibility vis# on whole methods = txns

Find visibility vis on method-internal events

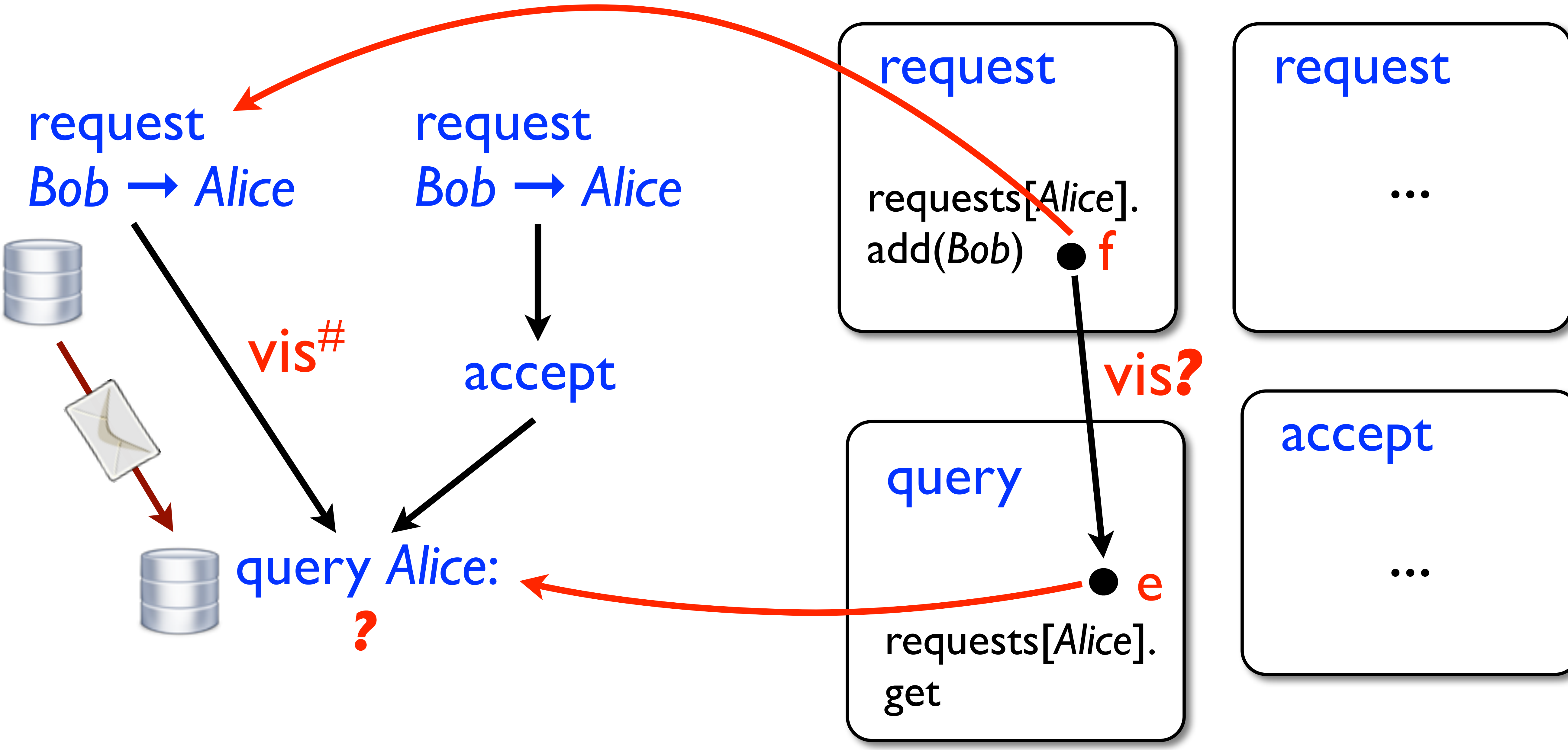
Coarse-grain context N ➜ Fine-grain execution ɣ(N)



Visibility = message delivery

All txn updates delivered together

Coarse-grain context N ➜ Fine-grain execution ɣ(N)



Visibility = message delivery

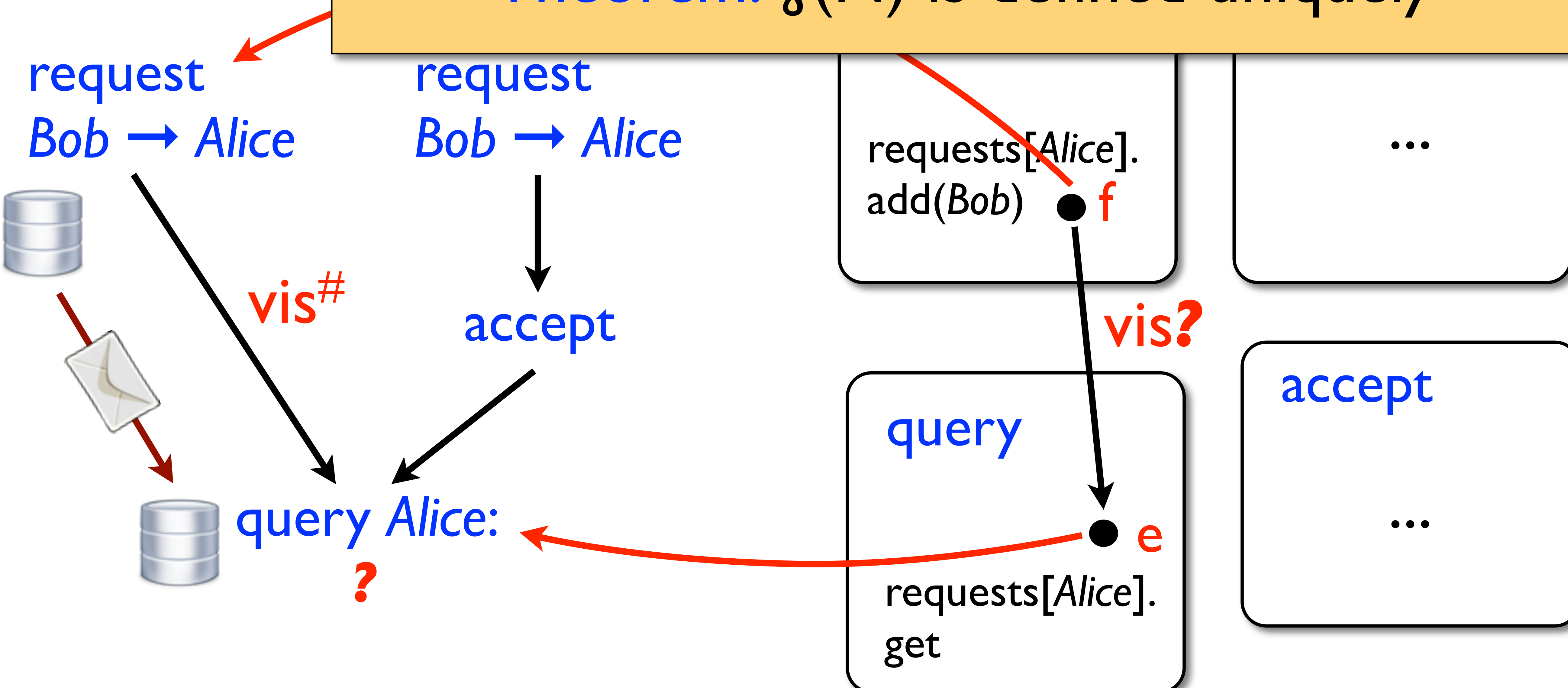
All txn updates delivered together

$(f, e) \in vis \Leftrightarrow (method(f), method(e)) \in vis^{\#}$

Coarse-grai

Only a constraint on fine-grain executions

Theorem: $\gamma(N)$ is defined uniquely

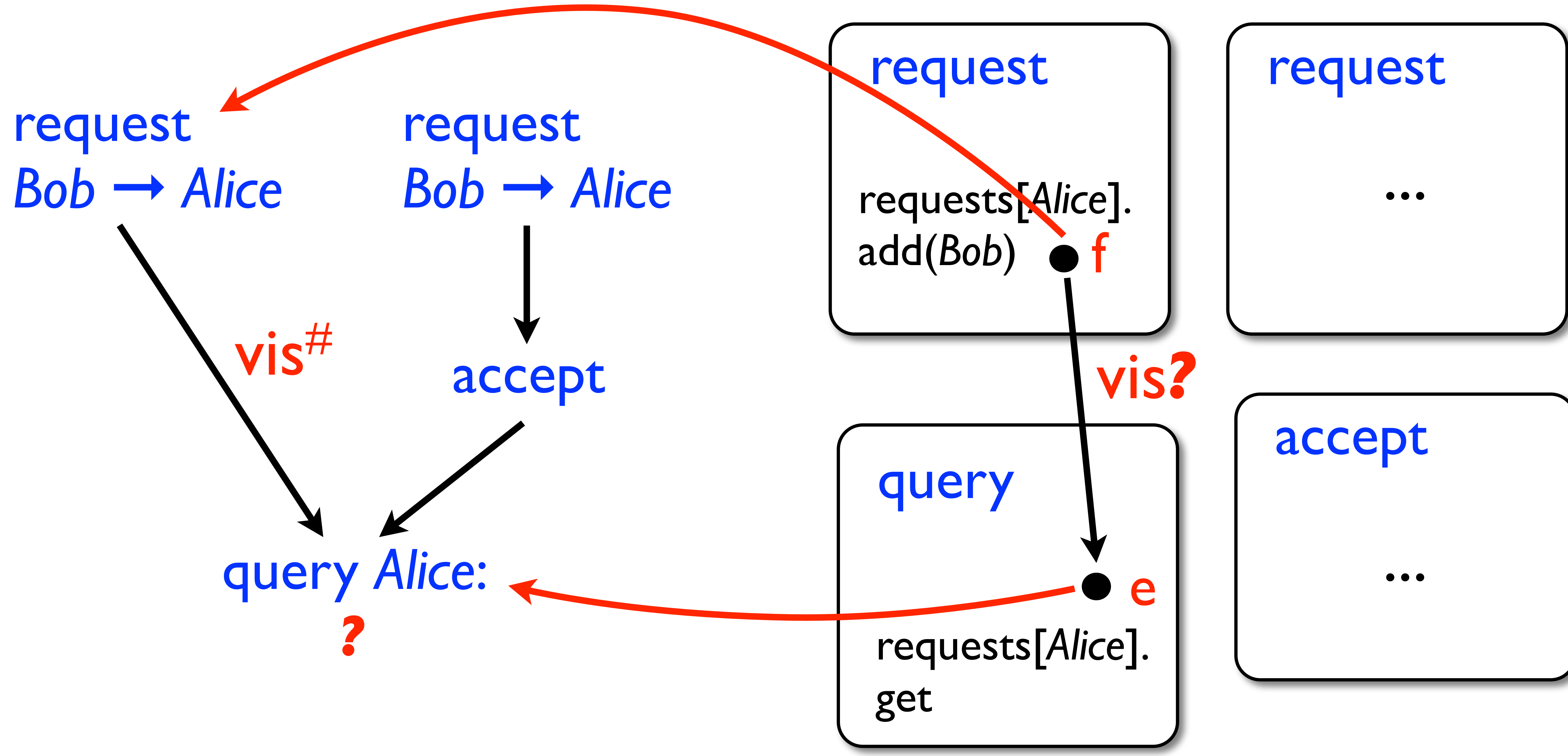


Visibility = message delivery

All txn updates delivered together

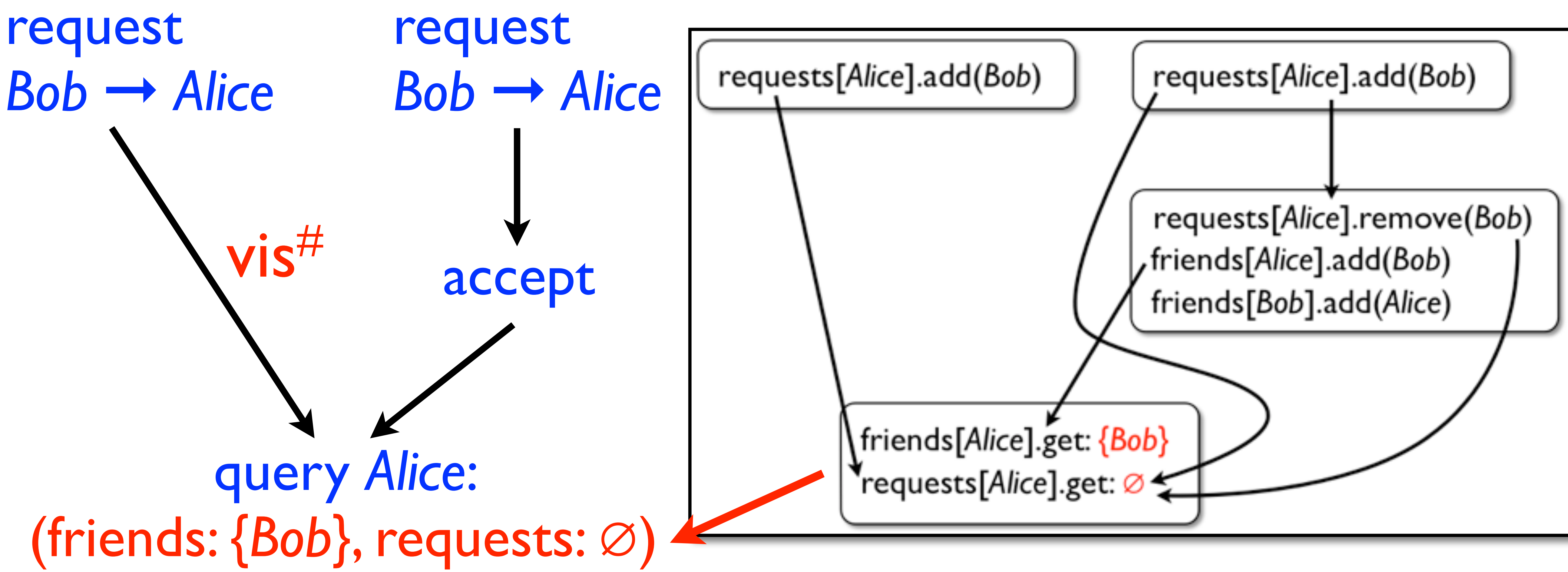
$(f, e) \in \text{vis} \Leftrightarrow (\text{method}(f), \text{method}(e)) \in \text{vis}^{\#}$

Coarse-grain context N ➜ Fine-grain execution ɣ(N)



To determine ⟦D⟧ on context N of operation op ➜ Determine the concretisation ɣ(N) wrt D ➜ Take the return value of op in ɣ(N)

Coarse-grain context N ➜ Fine-grain execution γ(N)



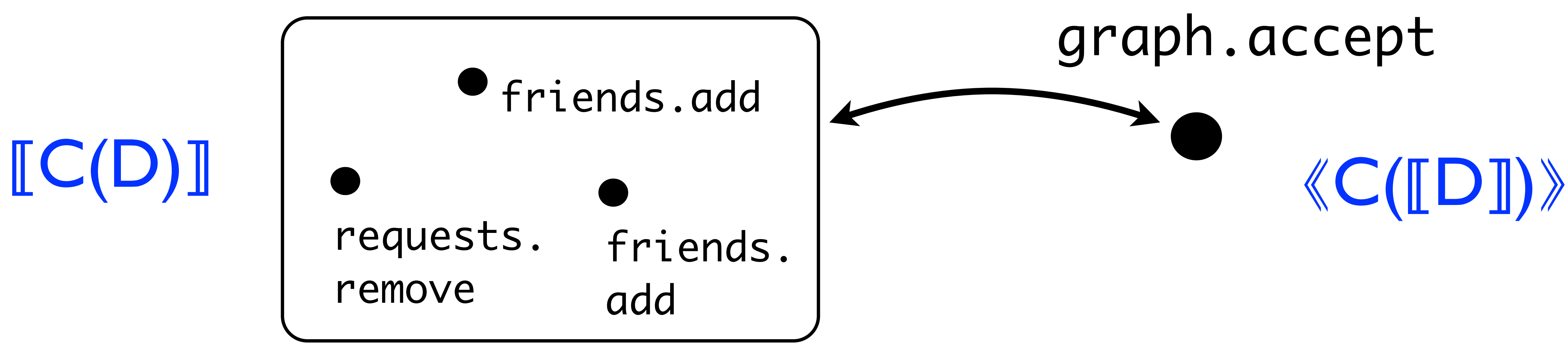
To determine ⟦D⟧ on context N of operation op ➜ Determine the concretisation γ(N) wrt D ➜ Take the return value of op in γ(N)

Theorem: $[\![C(D)]\!] = \langle\!\langle C([\![D]\!]) \rangle\!\rangle$

- Coarse-grain and fine-grain semantics coincide
- Client behaviour the same whether using implementation D or spec $[\![D]\!]$

Theorem: $[\![C(D)]\!] = \langle\!\langle C([\![D]\!]) \rangle\!\rangle$

- Consistency axioms are global: constrain the whole execution
- Preserve axioms when performing local execution transformations:



- 93-page extended version, enjoy!

Using the denotation

- ⟦D⟧ not effectively computable:

  1. Write a spec S describing desirable behaviour of D declaratively

  2. Prove S = ⟦D⟧

- Definition of ⟦D⟧ gives a proof method

- Examples: social graph, shopping cart

Conclusion

- Modular reasoning method for applications using eventually consistent databases: reflects application modularity

- Only causal consistency model: general theory?

- Opportunity: extend rich programming language theories to modern distributed systems