Scalable Termination Detection for Distributed Actor Systems

Dan Plyukhin and Gul Agha
UIUC

Actor Model

Actors are lightweight, stateful, async processes.

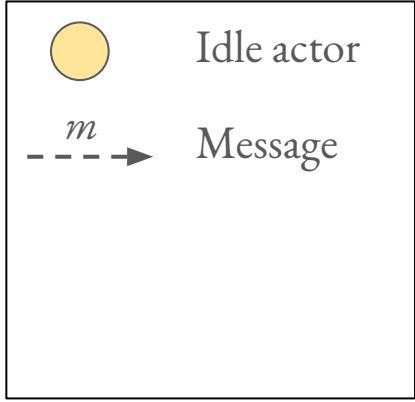
Used to build **low-latency distributed systems** (e.g. Riak, Discord, CouchDB).

Most popular frameworks (Erlang, Akka, Orleans) do not garbage collect actors.

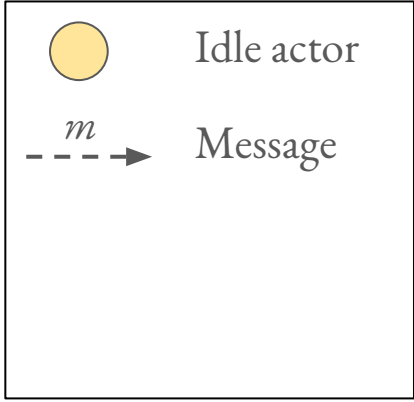
Ordinary tracing GC techniques don’t work.

Other solutions don’t scale well.

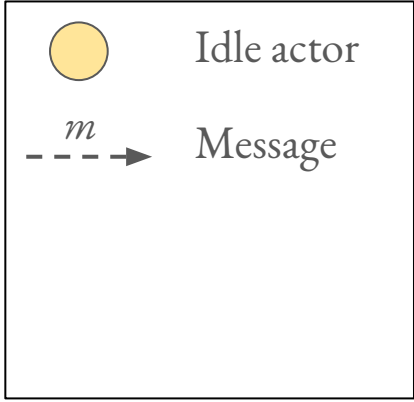
Part I: Actors


Idle actor
m
Message

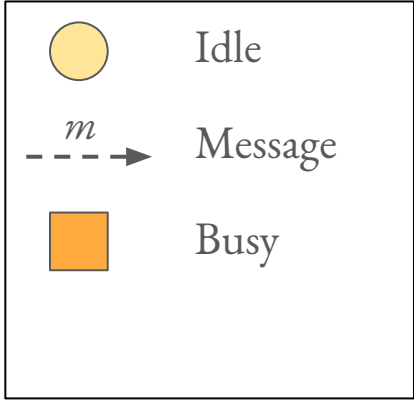

m


Idle actor
m
Message

Once an **idle** actor receives a message, it becomes **busy**.

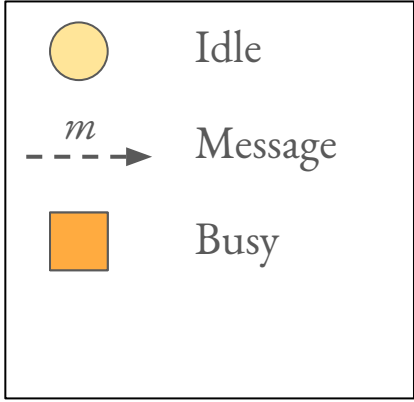


Once an **idle** actor receives a message, it becomes **busy**.



Once an **idle** actor receives a message, it becomes **busy**.

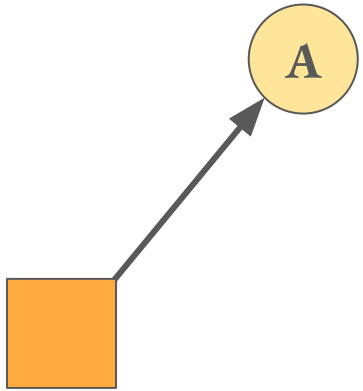
A busy actor can...

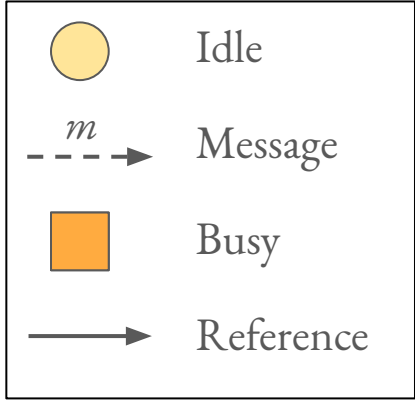


Once an **idle** actor receives a message, it becomes **busy**.

A busy actor can...

- spawn actors...

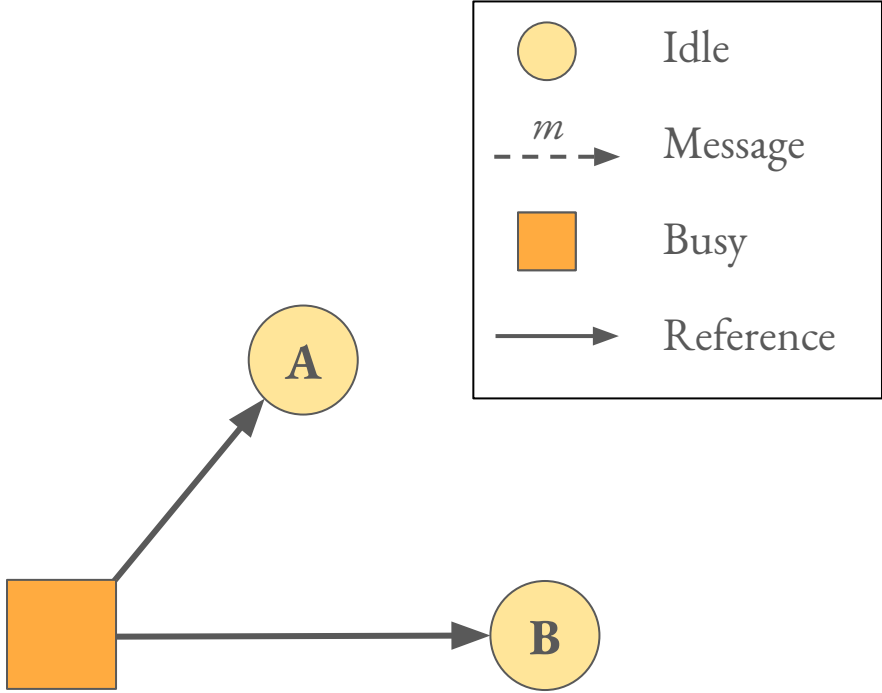




Once an **idle** actor receives a message, it becomes **busy**.

A busy actor can...

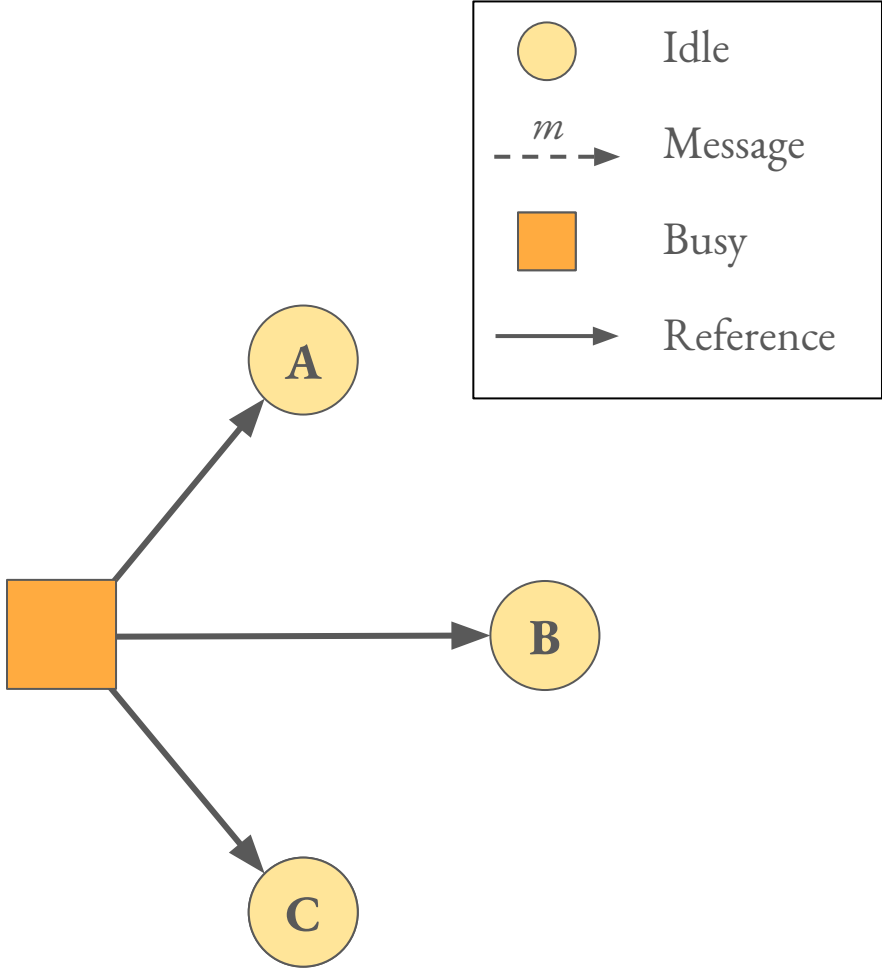
- spawn actors...



Once an **idle** actor receives a message, it becomes **busy**.

A busy actor can...

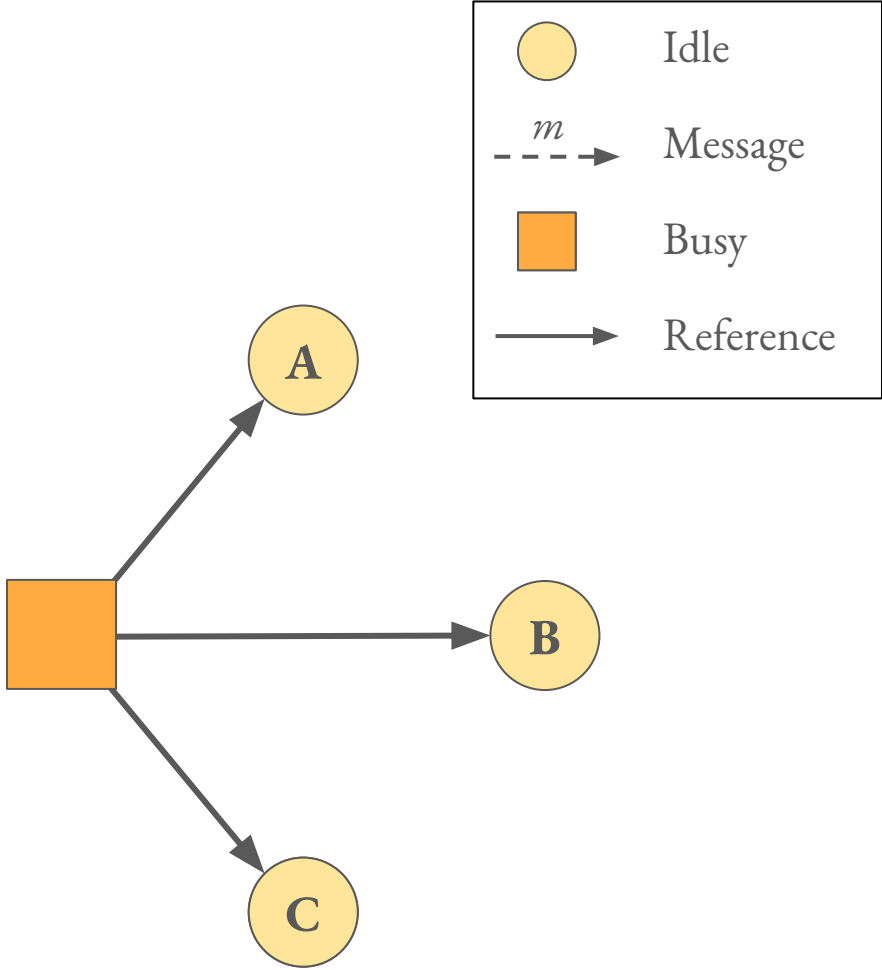
- spawn actors...



Once an **idle** actor receives a message, it becomes **busy**.

A busy actor can...

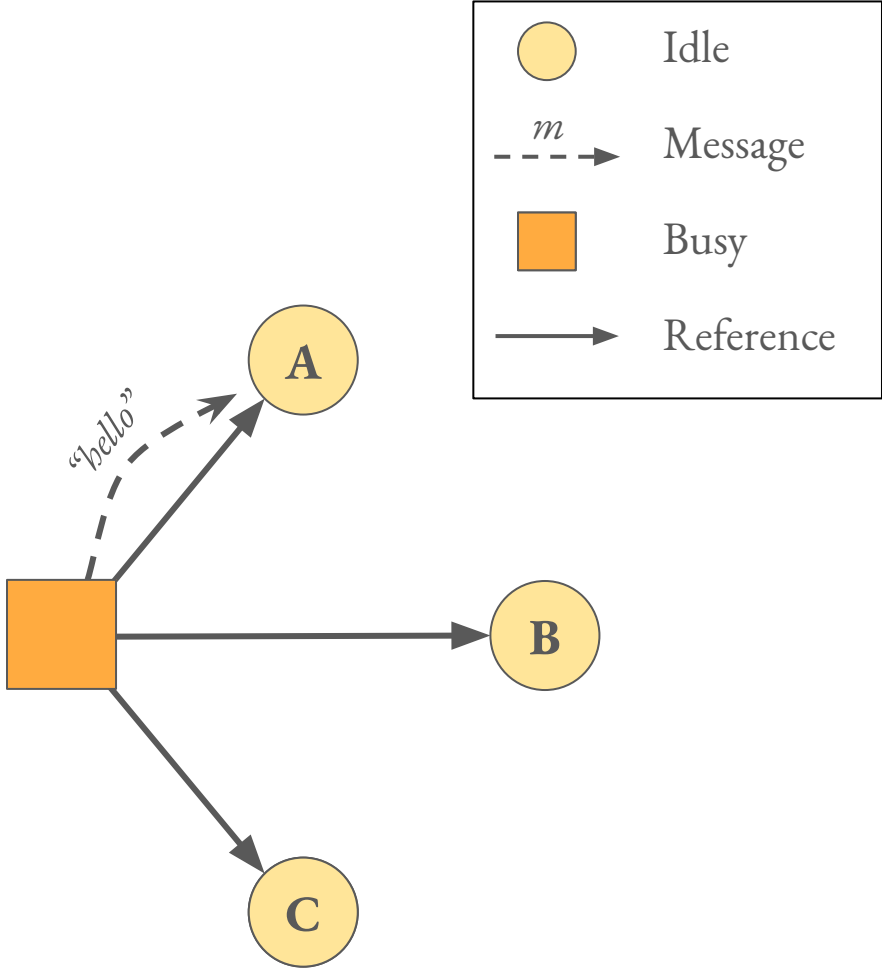
- spawn actors...
- send async messages...



Once an **idle** actor receives a message, it becomes **busy**.

A busy actor can...

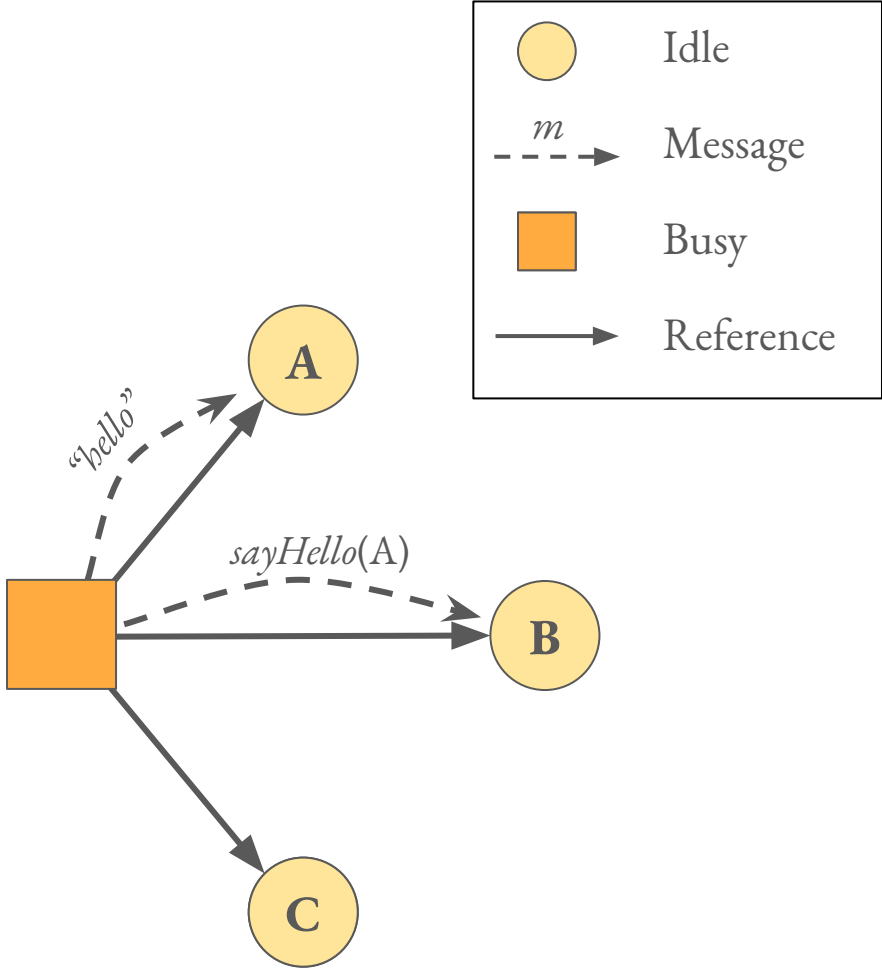
- spawn actors...
- send async messages...



Once an **idle** actor receives a message, it becomes **busy**.

A busy actor can...

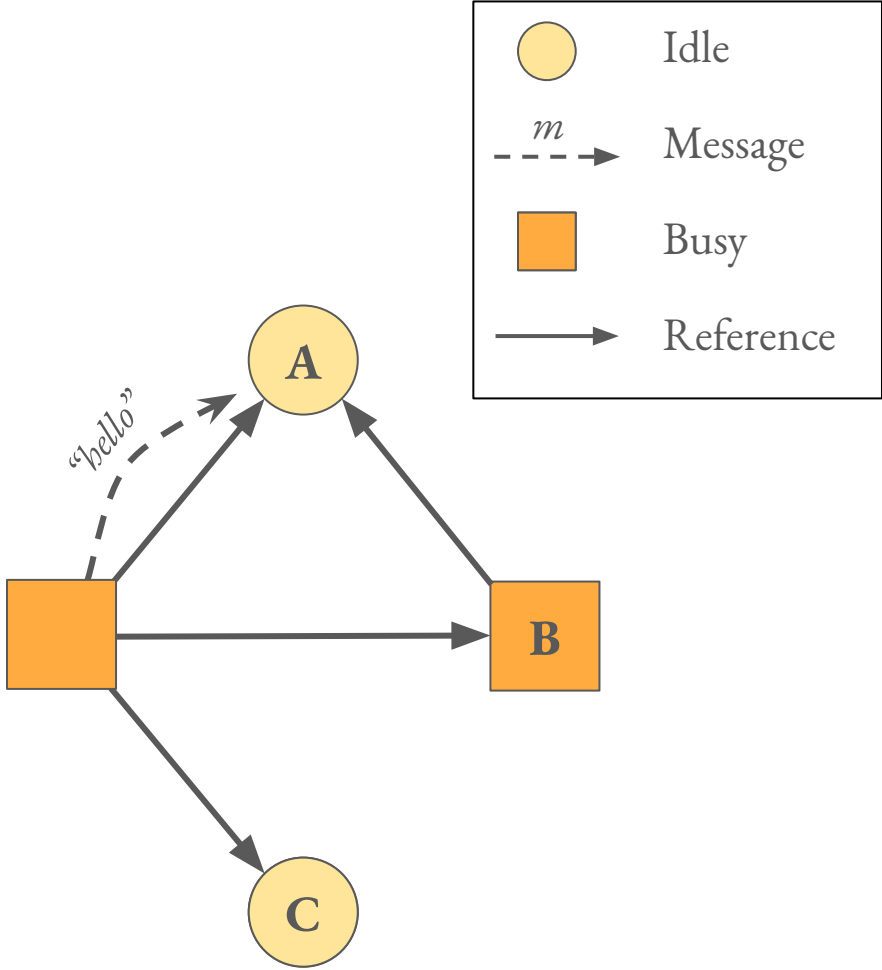
- spawn actors...
- send async messages...



Once an **idle** actor receives a message, it becomes **busy**.

A busy actor can...

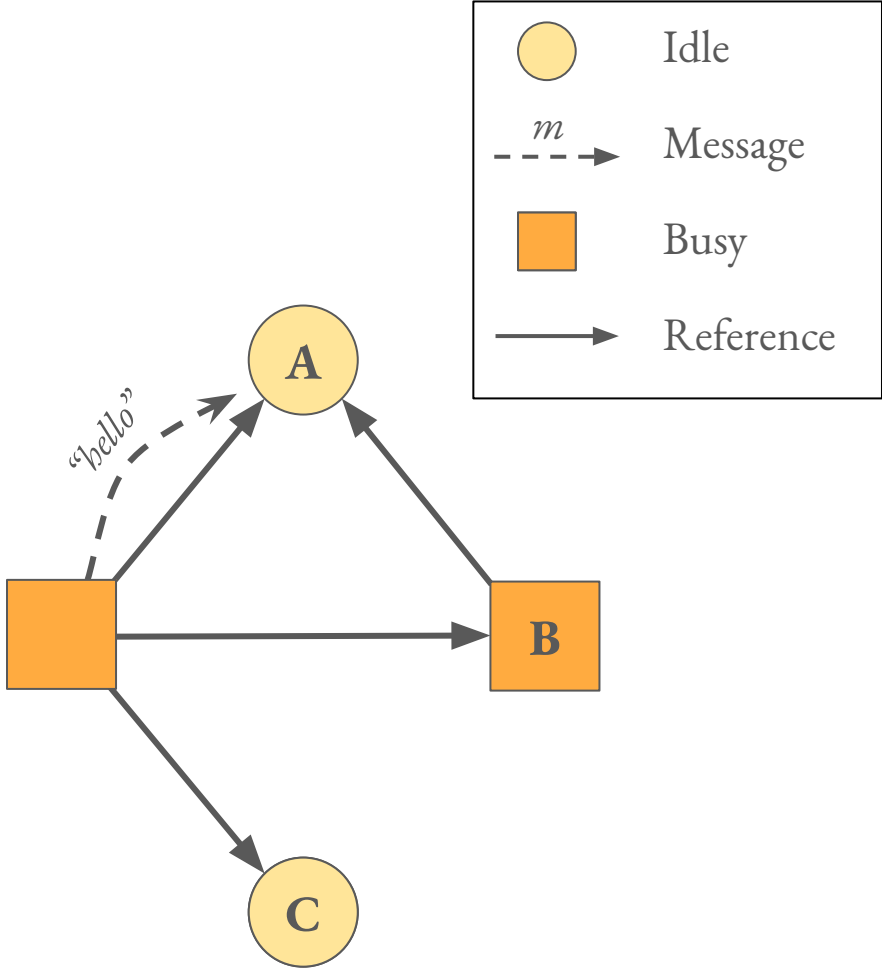
- spawn actors...
- send async messages...



Once an **idle** actor receives a message, it becomes **busy**.

A busy actor can...

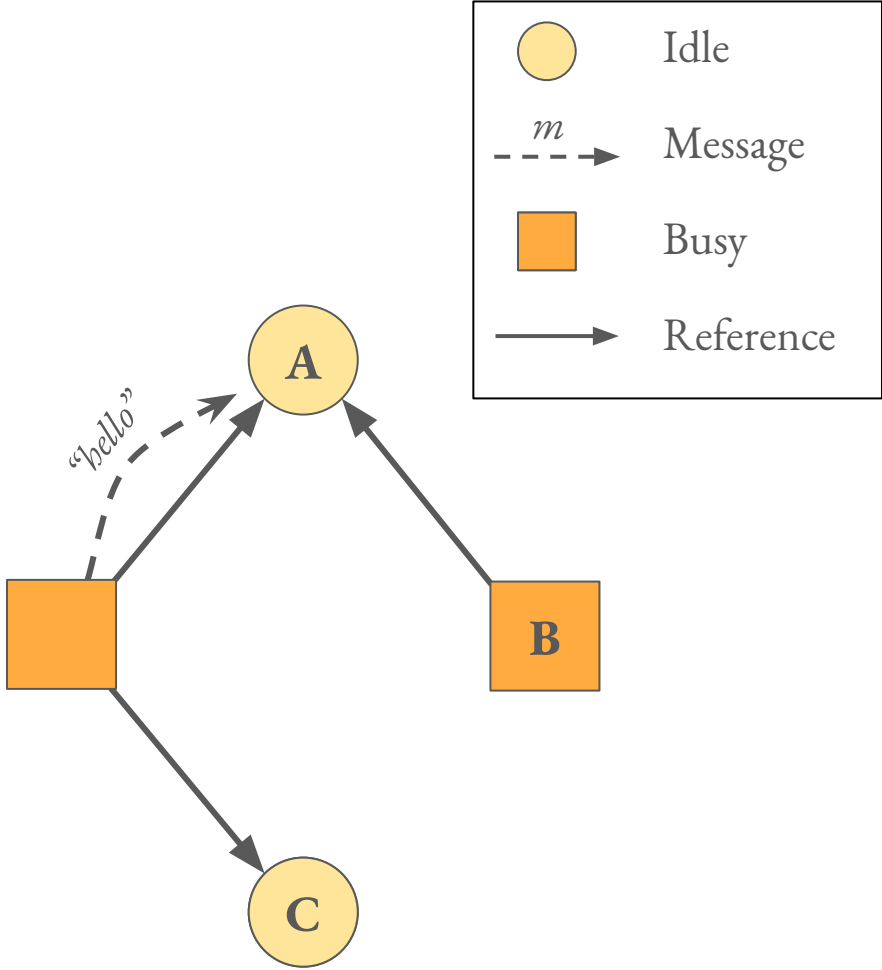
- spawn actors...
- send async messages...
- update its local state...



Once an **idle** actor receives a message, it becomes **busy**.

A busy actor can...

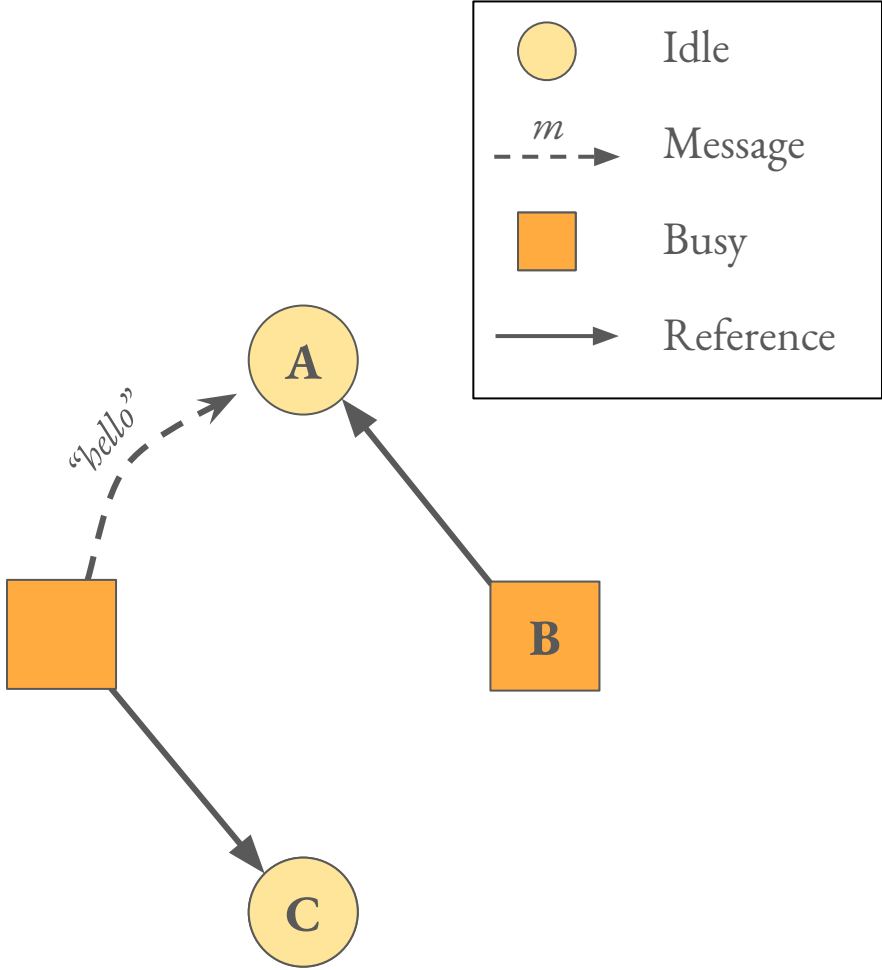
- spawn actors...
- send async messages...
- update its local state...



Once an **idle** actor receives a message, it becomes **busy**.

A busy actor can...

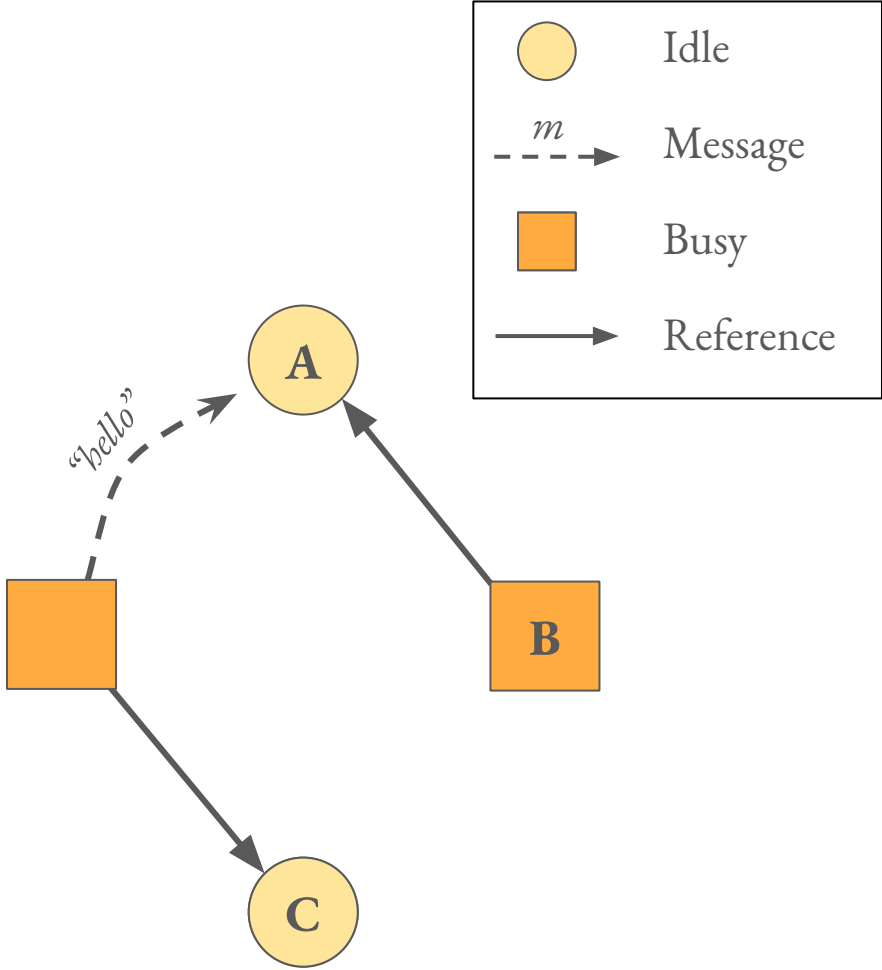
- spawn actors...
- send async messages...
- update its local state...



Once an **idle** actor receives a message, it becomes **busy**.

A busy actor can...

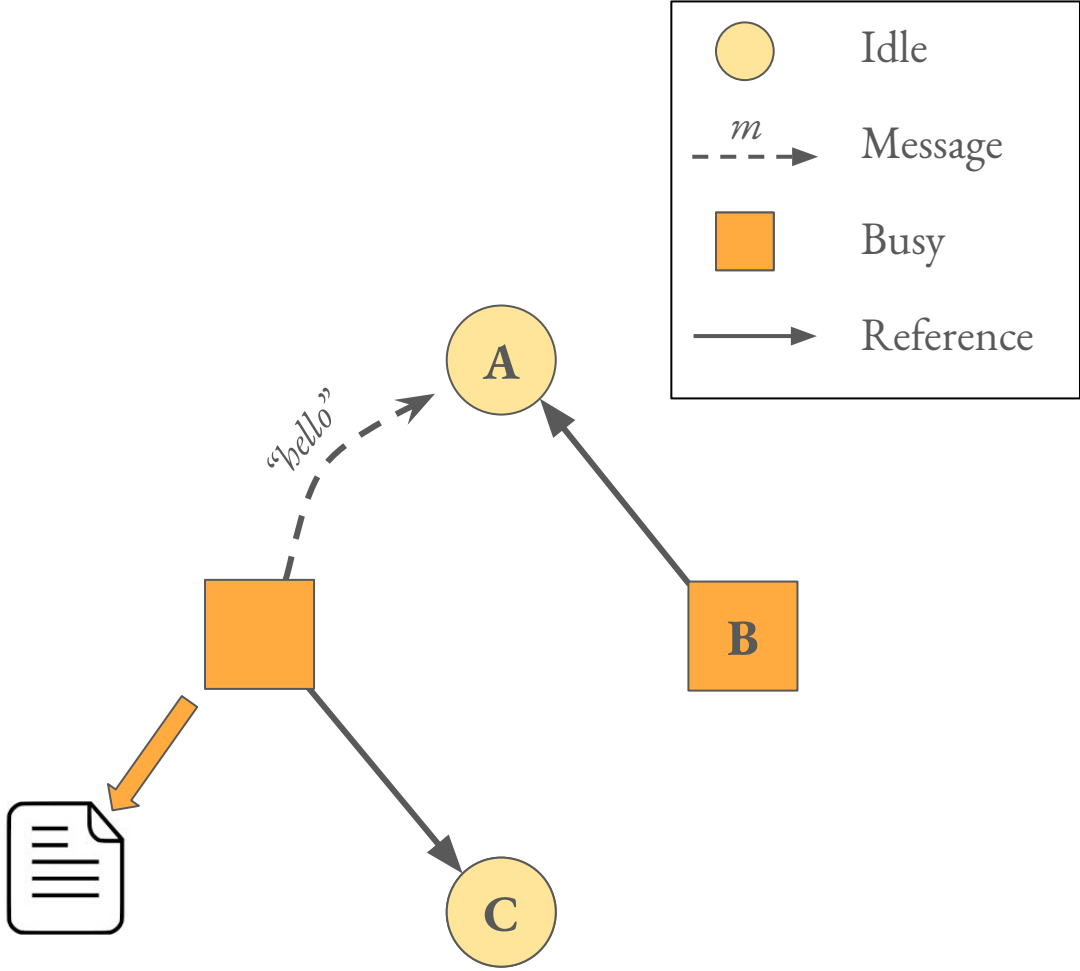
- spawn actors...
- send async messages...
- update its local state...
- perform effects...



Once an **idle** actor receives a message, it becomes **busy**.

A busy actor can...

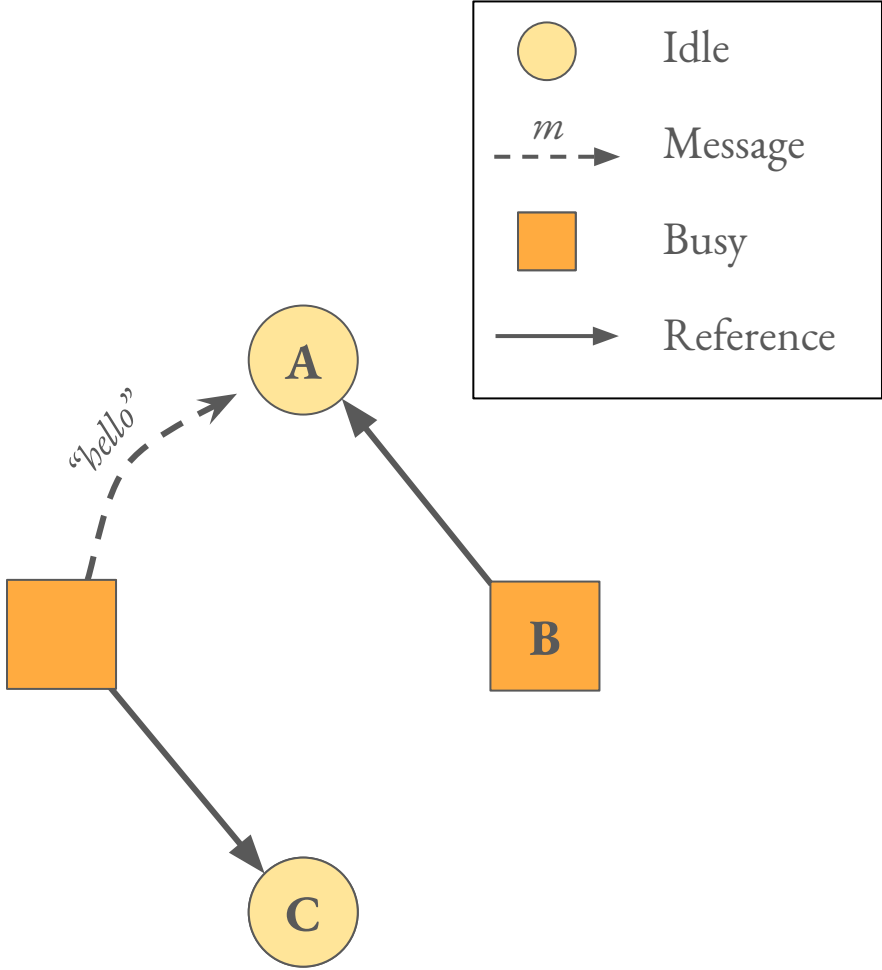
- spawn actors...
- send async messages...
- update its local state...
- perform effects...



Once an **idle** actor receives a message, it becomes **busy**.

A busy actor can...

- spawn actors...
- send async messages...
- update its local state...
- perform effects...

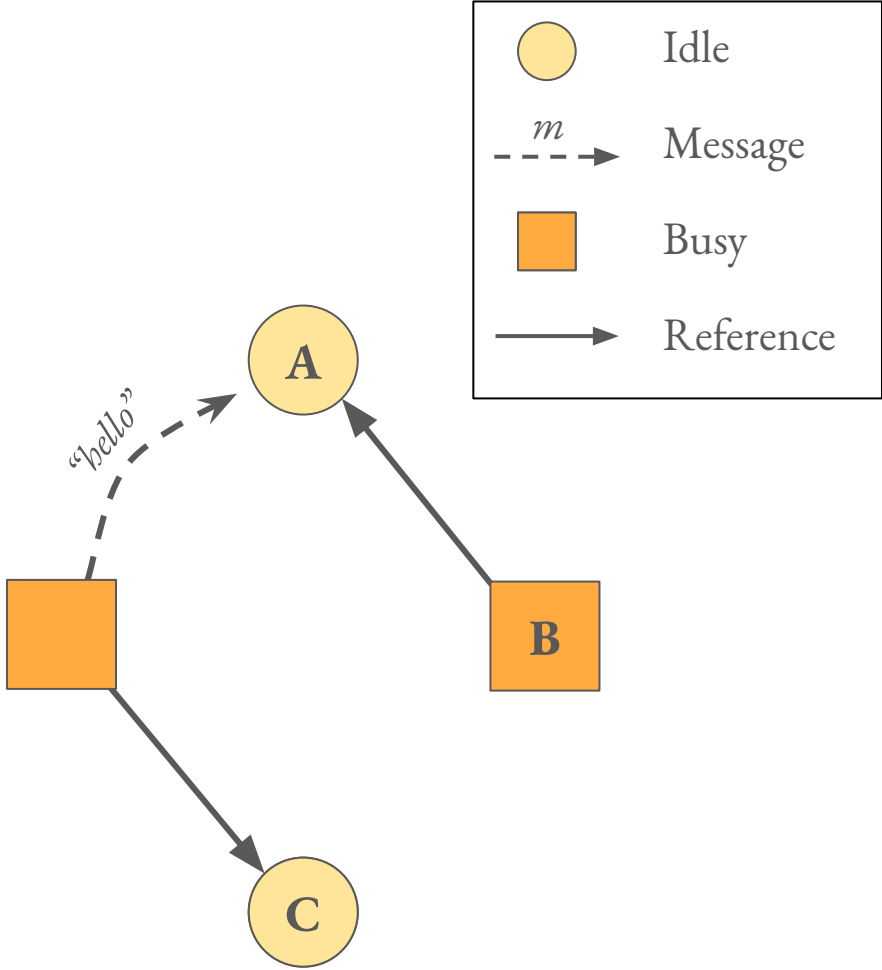


Once an **idle** actor receives a message, it becomes **busy**.

A busy actor can...

- spawn actors...
- send async messages...
- update its local state...
- perform effects...

...before becoming idle again.

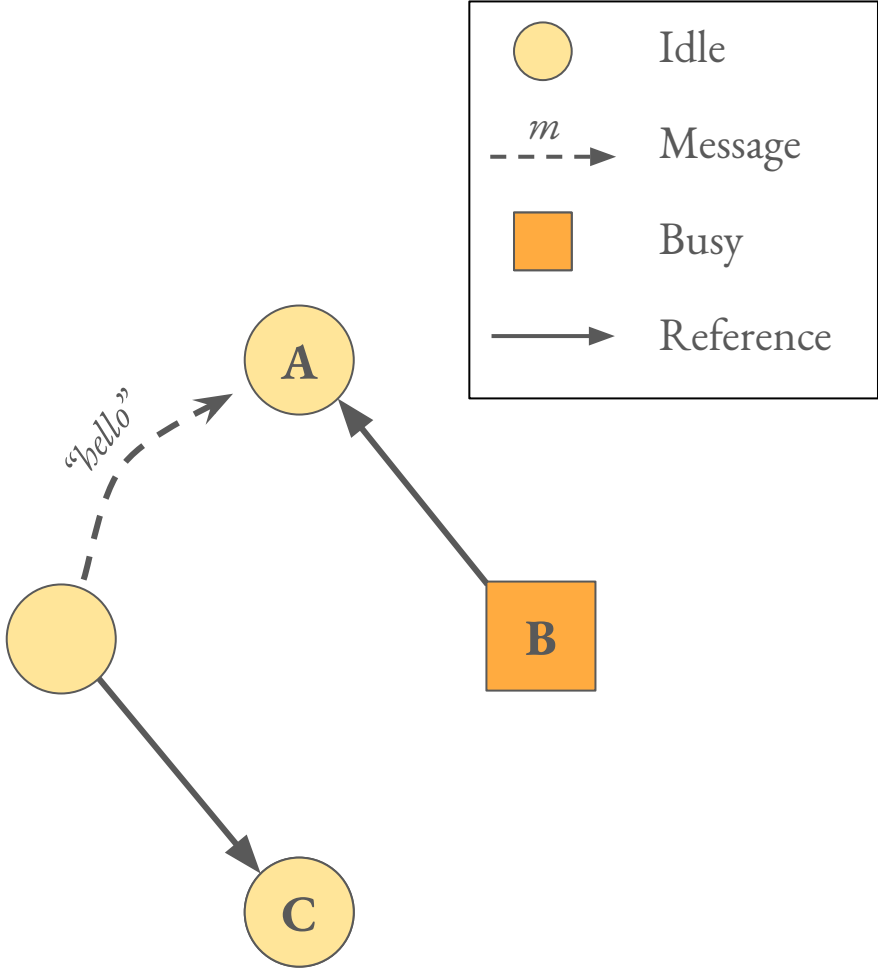


Once an **idle** actor receives a message, it becomes **busy**.

A busy actor can...

- spawn actors...
- send async messages...
- update its local state...
- perform effects...

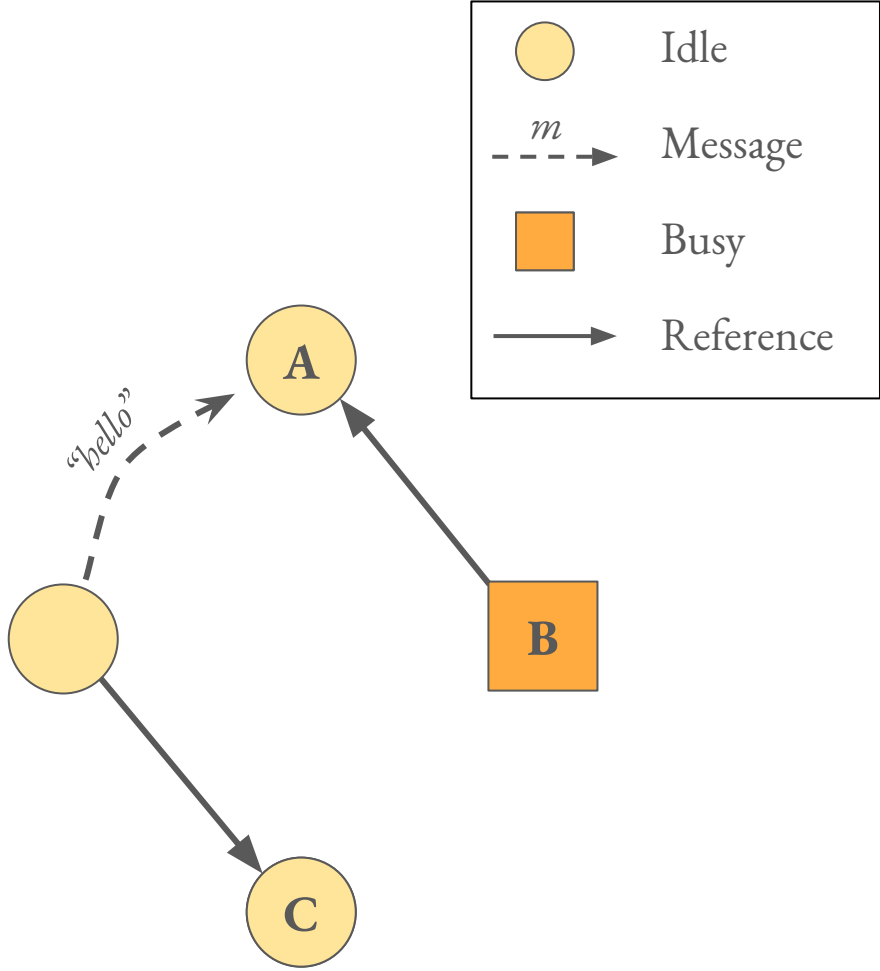
...before becoming idle again.



An actor is **unblocked** if:

a. it is busy, or

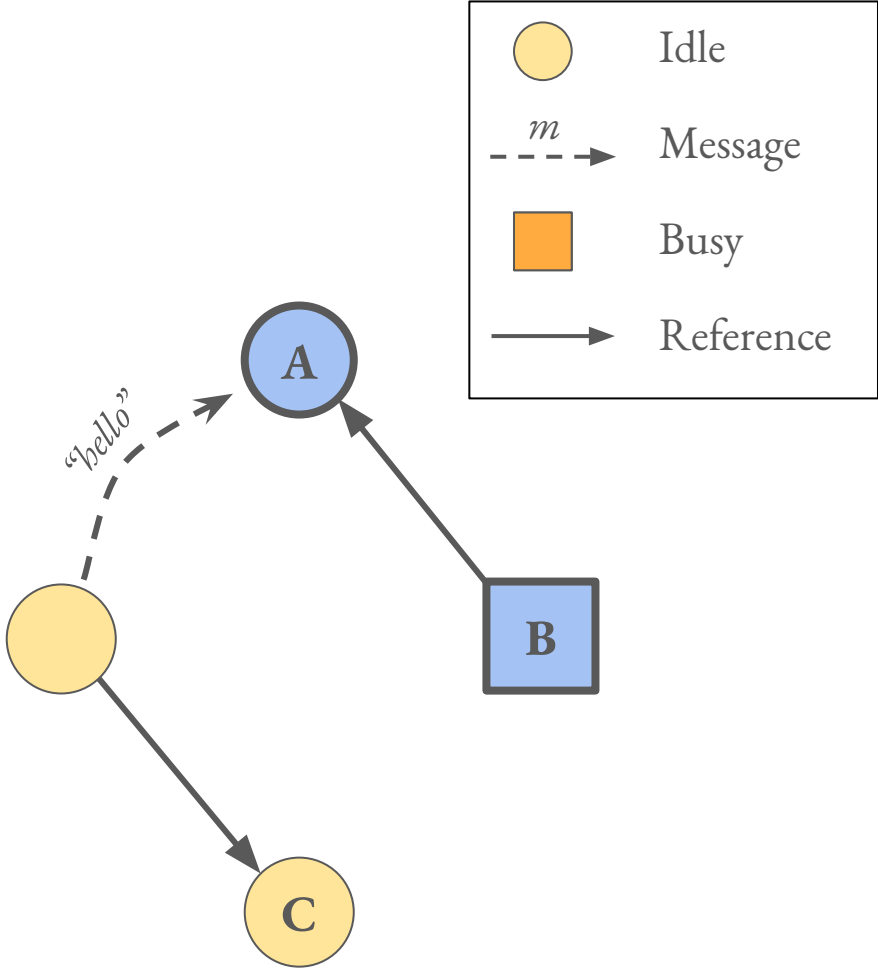
b. it has undelivered messages.



An actor is **unblocked** if:

a. it is busy, or

b. it has undelivered messages.

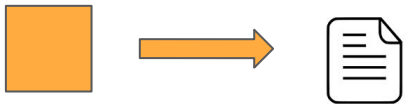


Part II: Garbage

An actor is **garbage** if it can be destroyed without affecting system behavior.

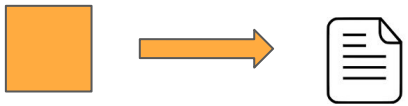
An actor is **garbage** if it can be destroyed without affecting system behavior.

If an actor could become unblocked, we shouldn't collect it.

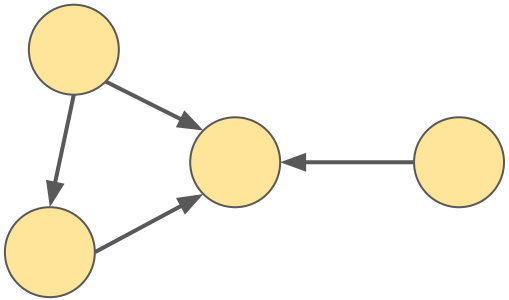


An actor is **garbage** if it can be destroyed without affecting system behavior.

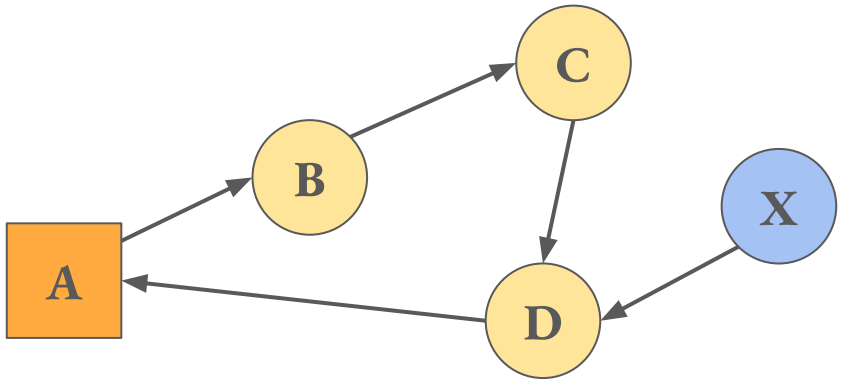
If an actor could become unblocked, we shouldn't collect it.



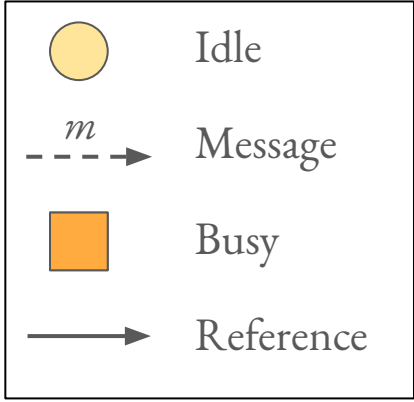
We want actors that are **permanently blocked**, aka **terminated** actors



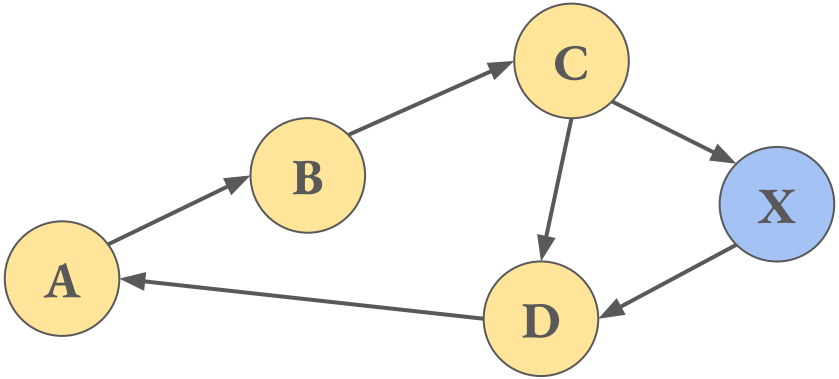
Is X terminated?



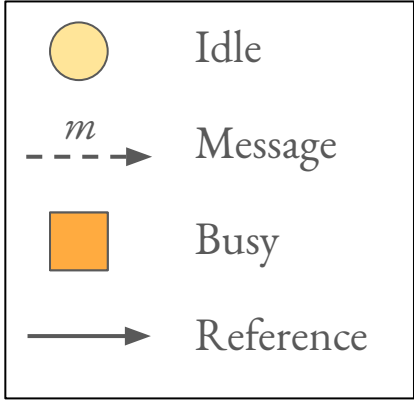
Yes.



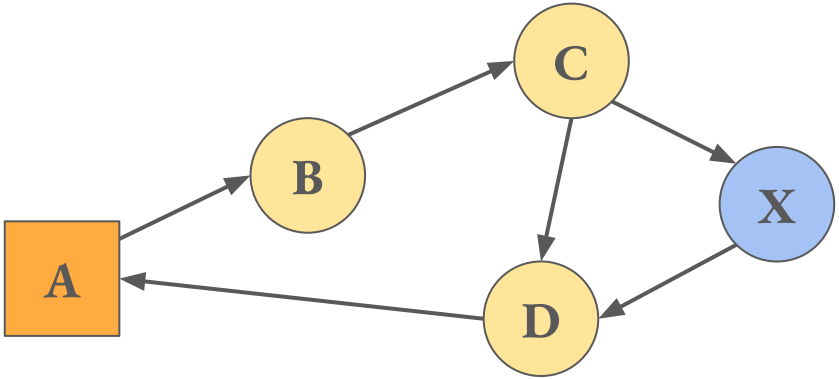
Is X terminated?

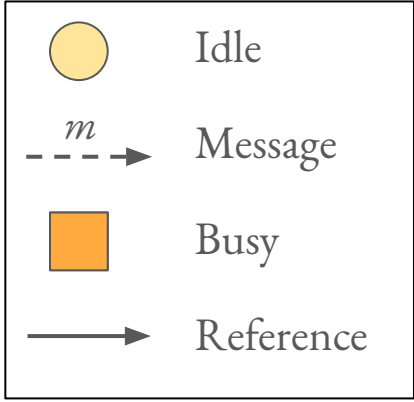


Yes.



Is X terminated?





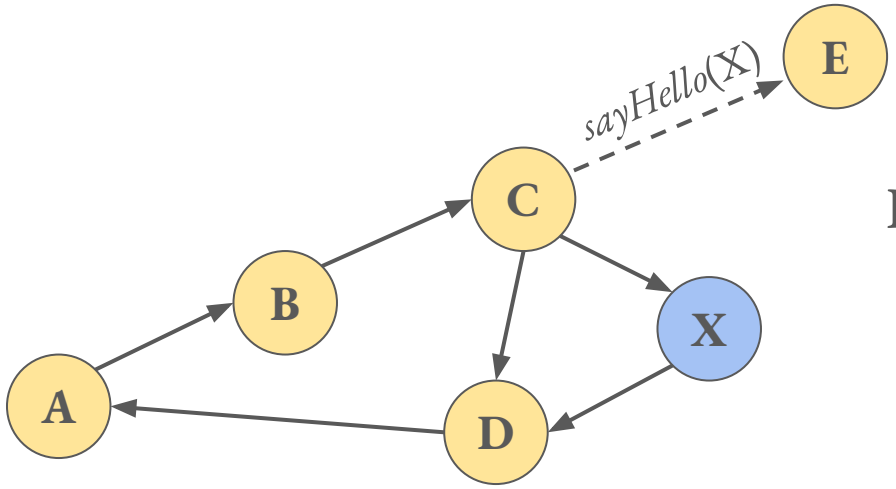
Possibly not!

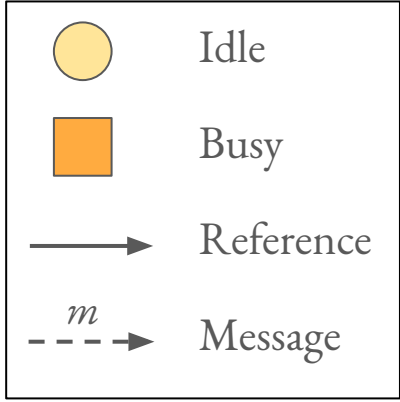
So X is terminated if all actors that can “reach” it are blocked?

So X is terminated if all actors that can “reach” it are blocked?

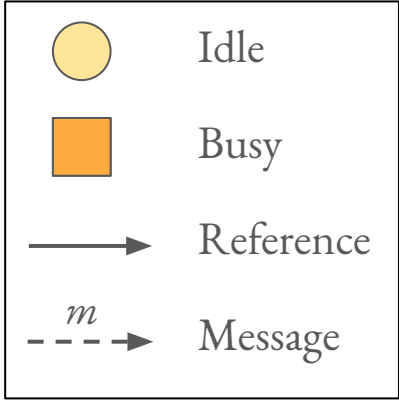
Not so fast!

Is X terminated?





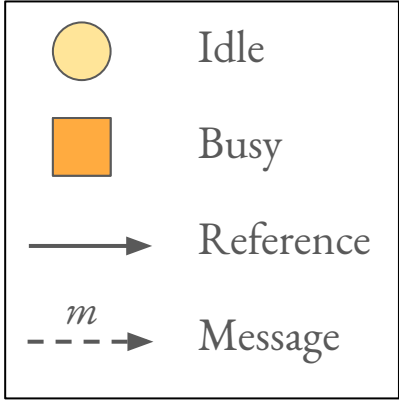
Possibly not.



A is **potentially acquainted** with B if either:

a. A has a reference to B

b. there is an undelivered message to A that contains a reference to B



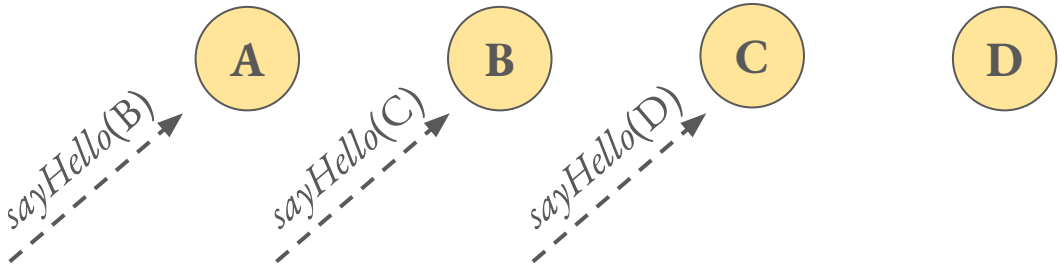
A is **potentially acquainted** with B if either:

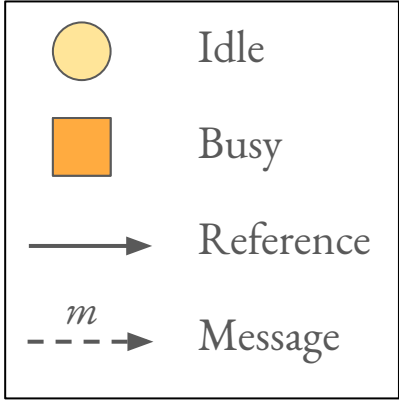
a. A has a reference to B

b. there is an undelivered message to A that contains a reference to B

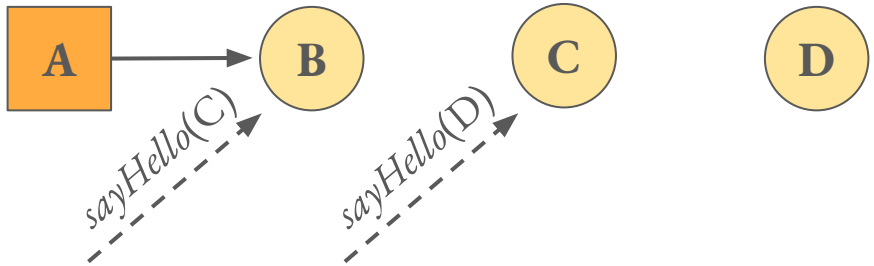
A is called a **potential inverse acquaintance** of B

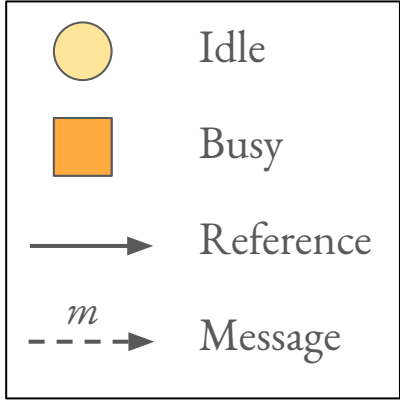
Potential reachability is the reflexive transitive closure of the potential acquaintance relation



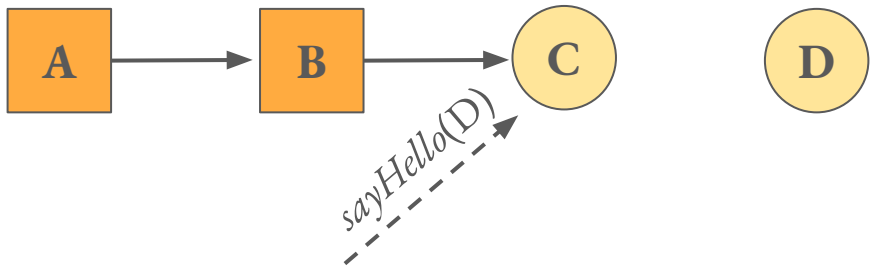


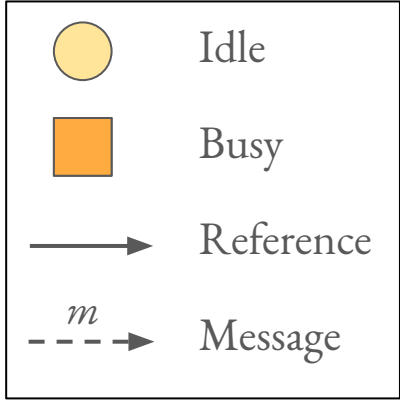
Potential reachability is the reflexive transitive closure of the potential acquaintance relation



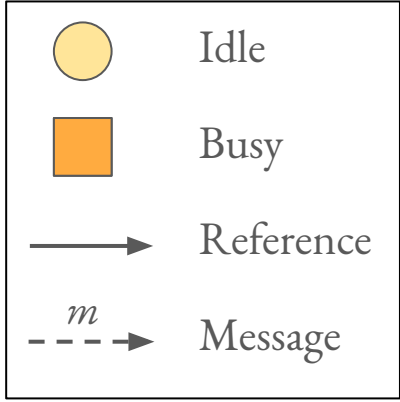


Potential reachability is the reflexive transitive closure of the potential acquaintance relation



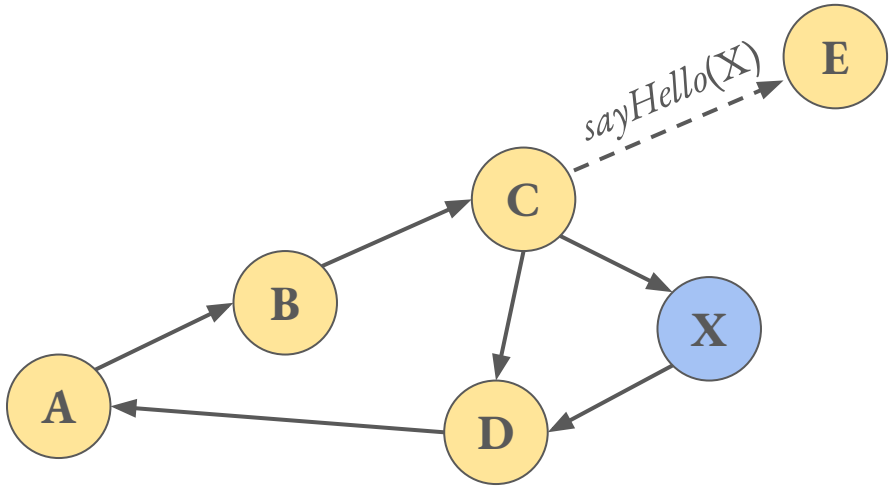


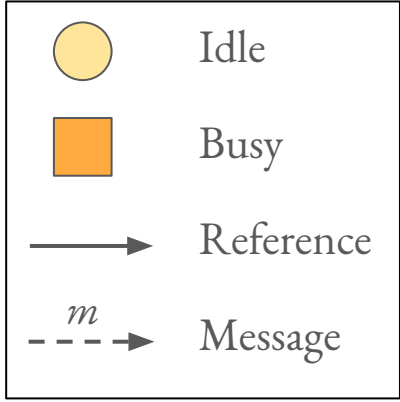
Potential reachability is the reflexive transitive closure of the potential acquaintance relation



X may not be terminated if it is

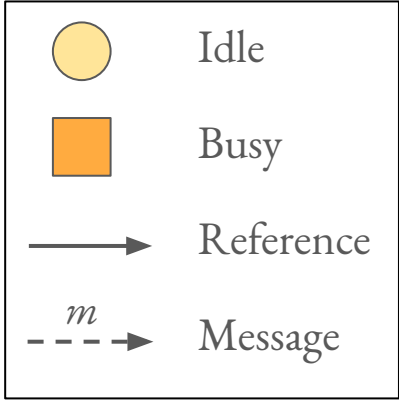
potentially reachable by an **unblocked actor**

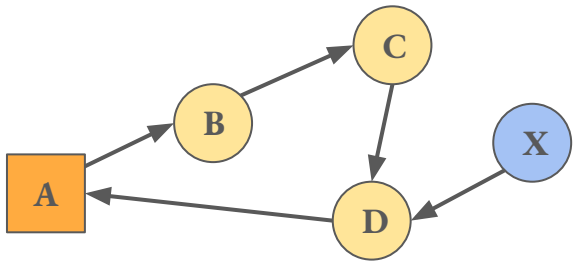


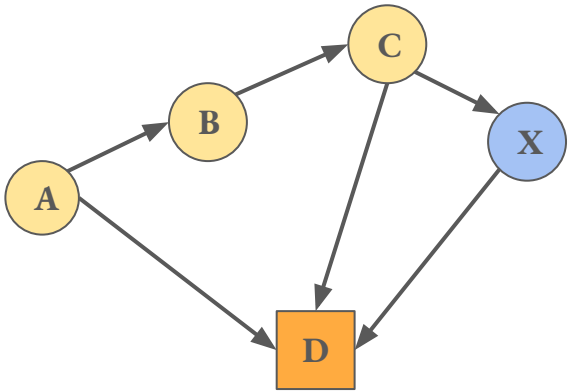


X is terminated if it is

potentially reachable only by **blocked actors**

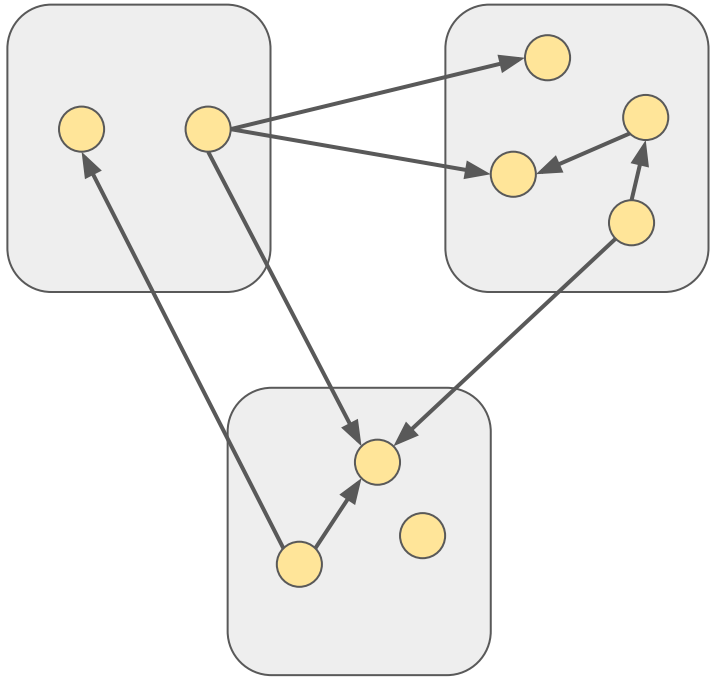




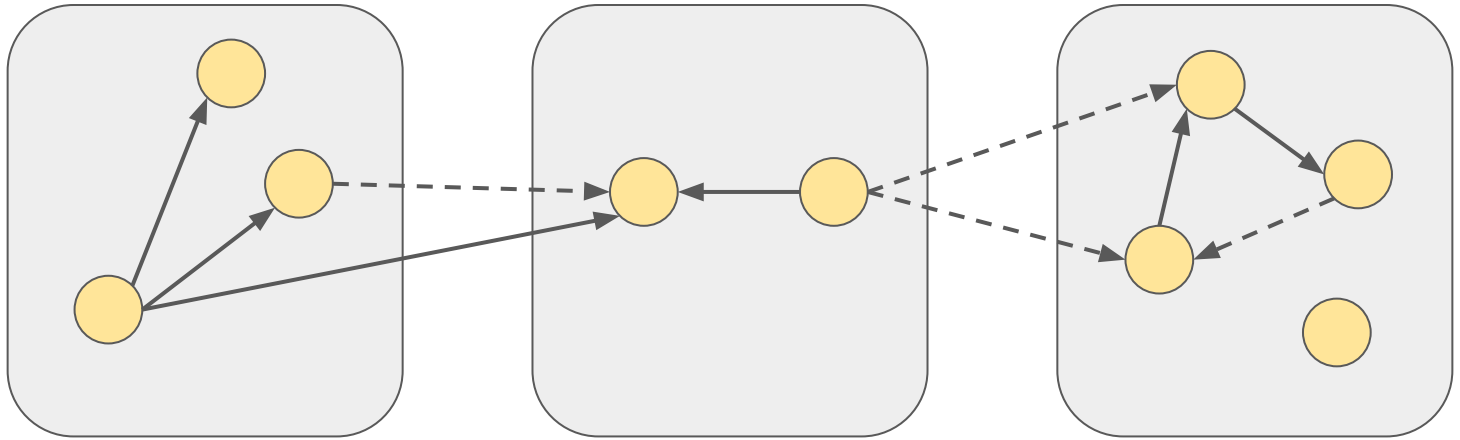


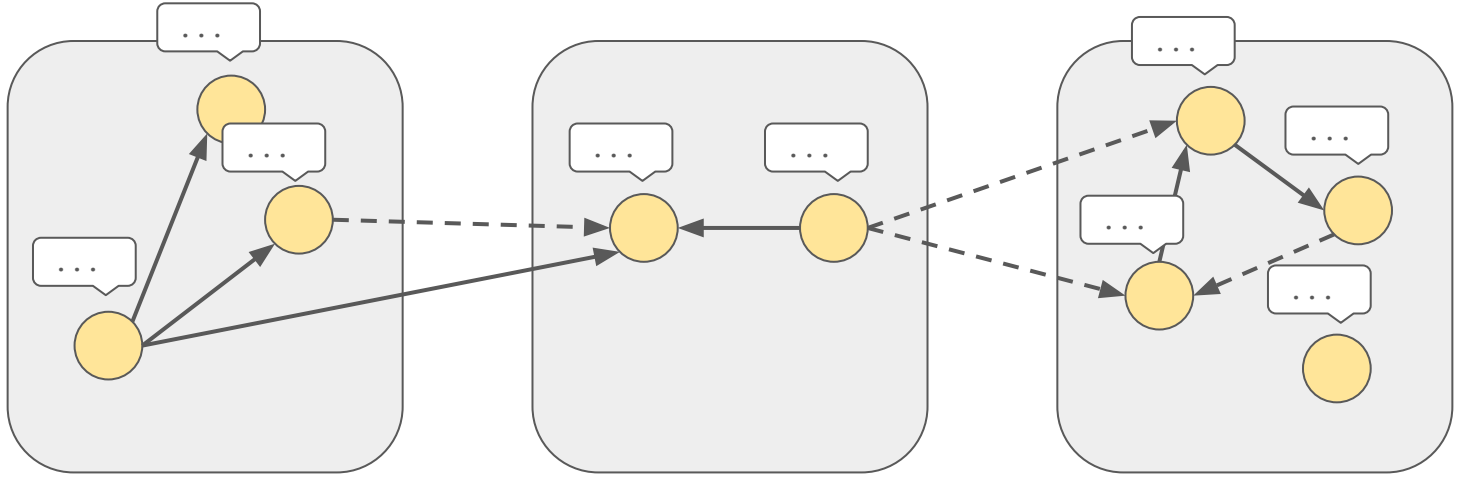
Related Work

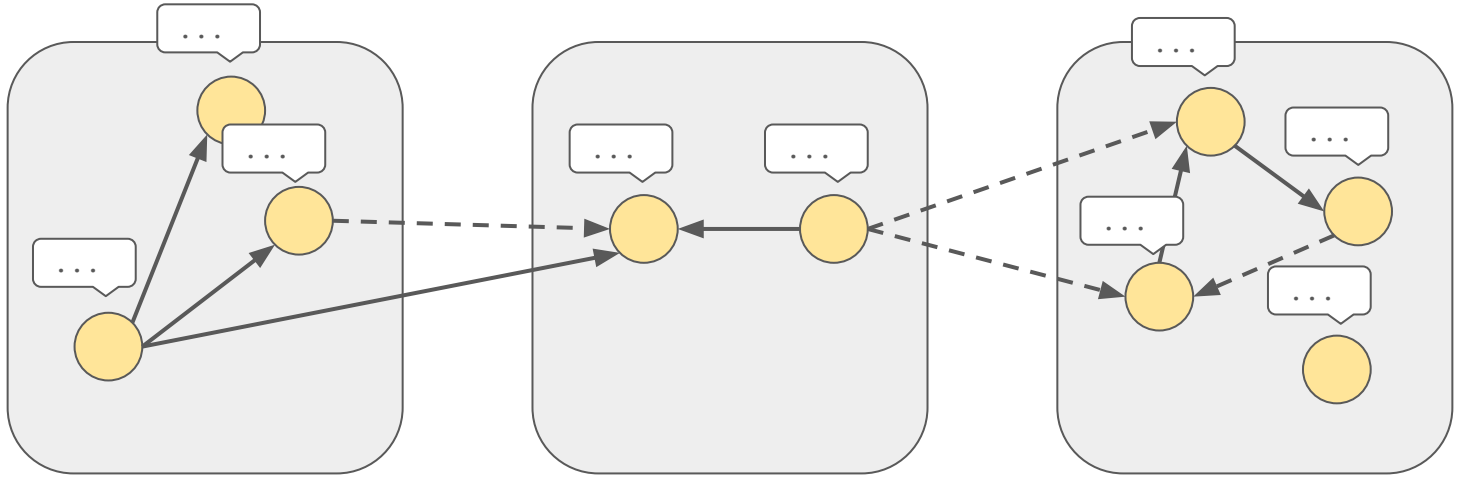
- Global snapshots
  - Not incremental
- SALSA: based on approximate snapshots
  - High overhead
- Pony: inspiration for this work
  - Causal message delivery is expensive

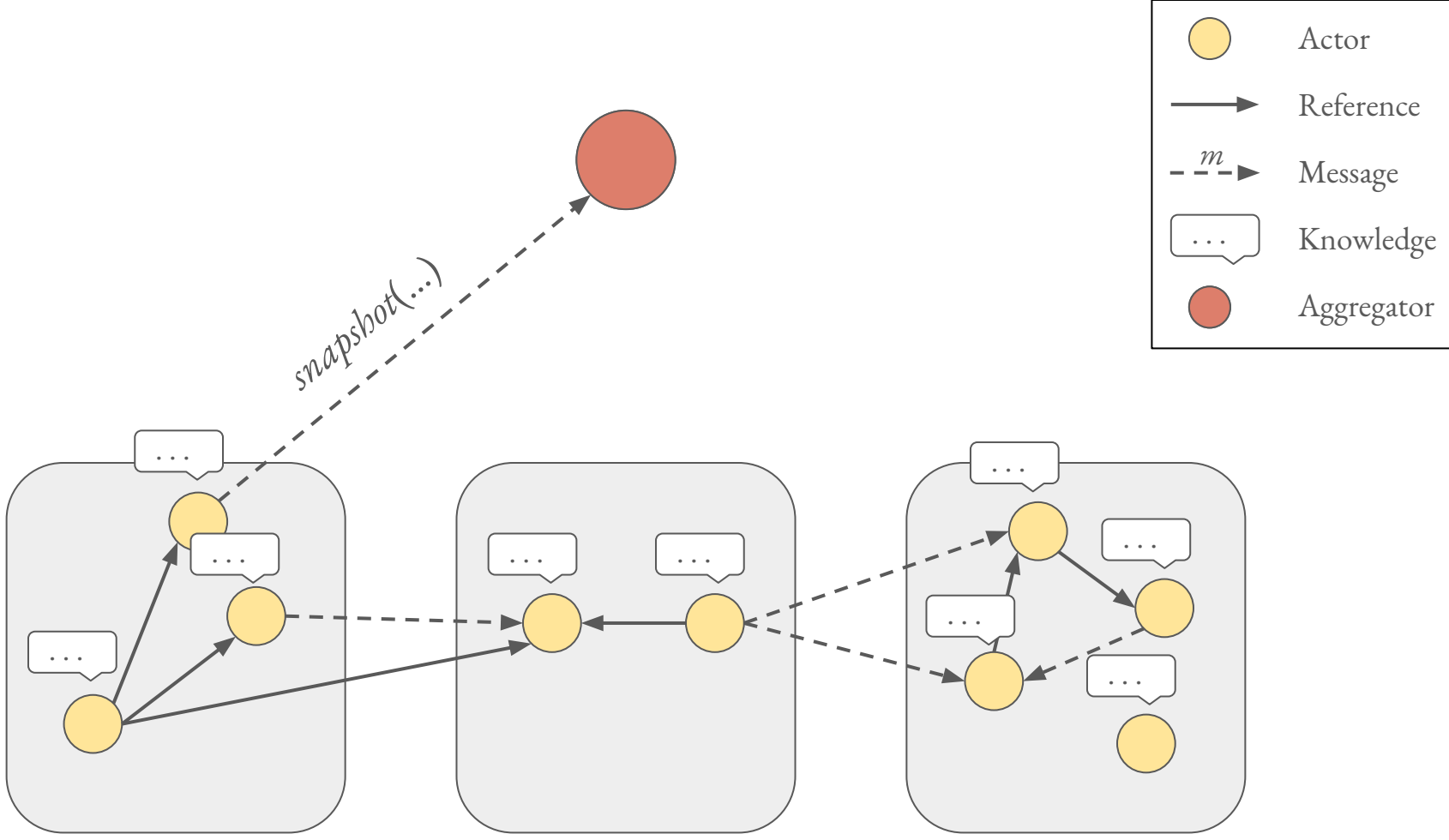


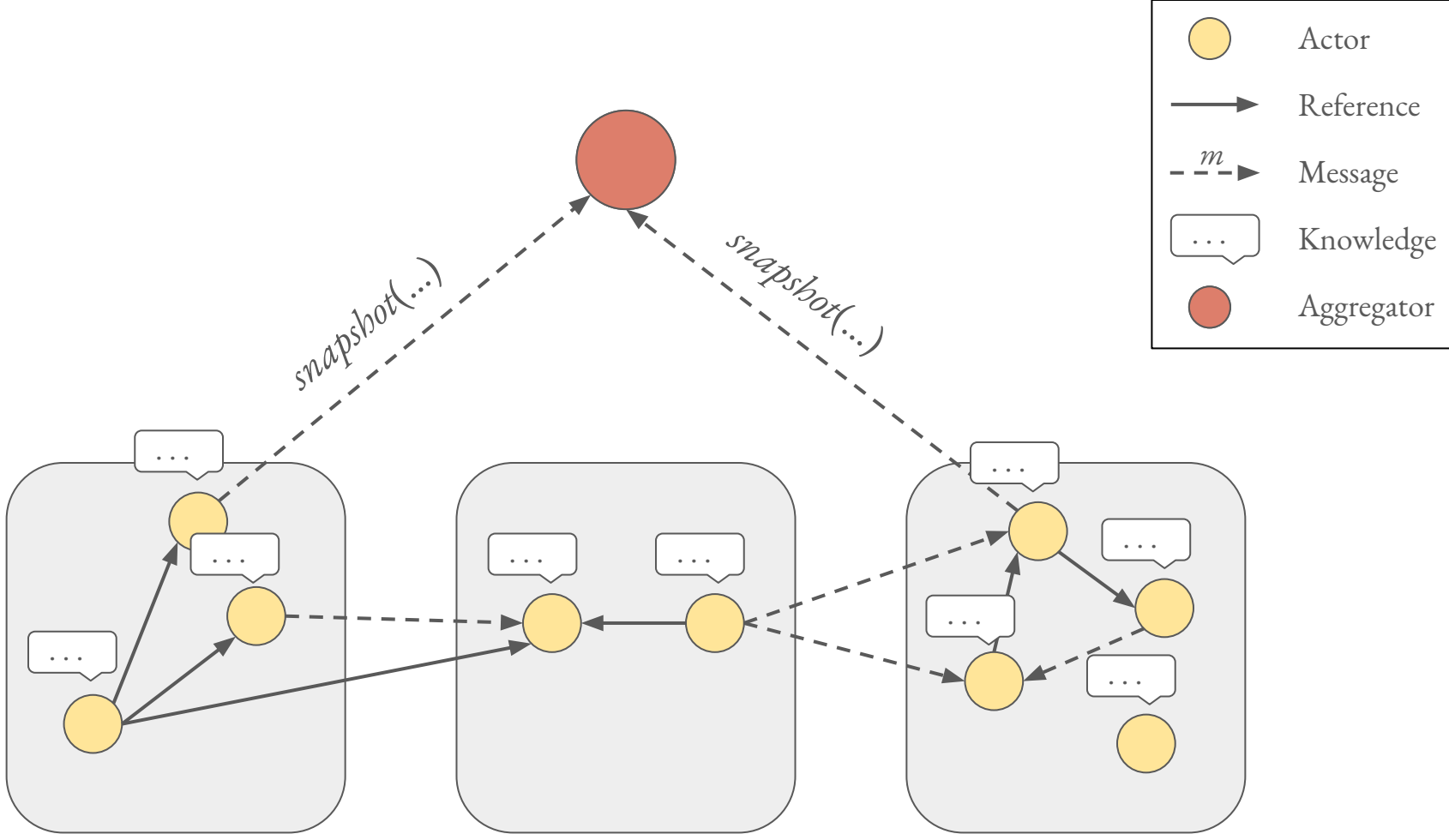
Demo

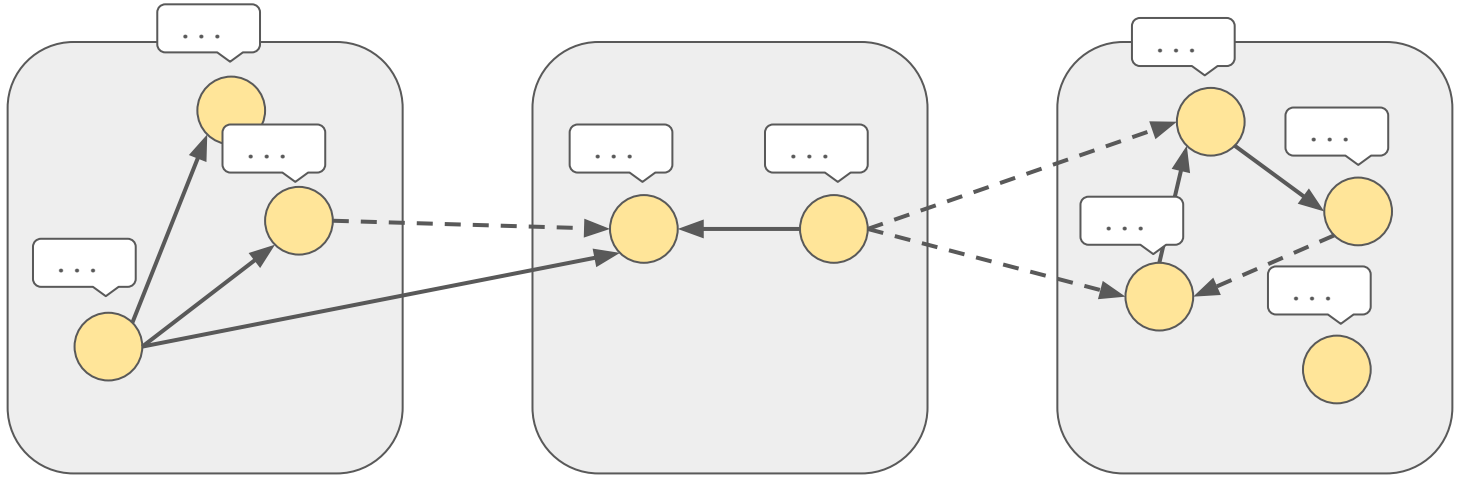

Actor
Reference
m
Message

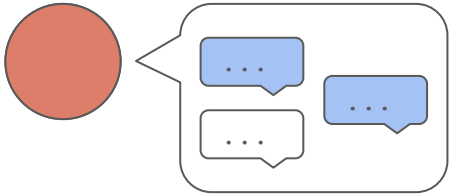

Actor
Reference
m
Message
Knowledge

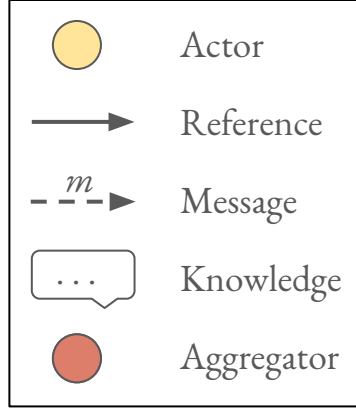

Actor
Reference
m
Message
Knowledge
Aggregator

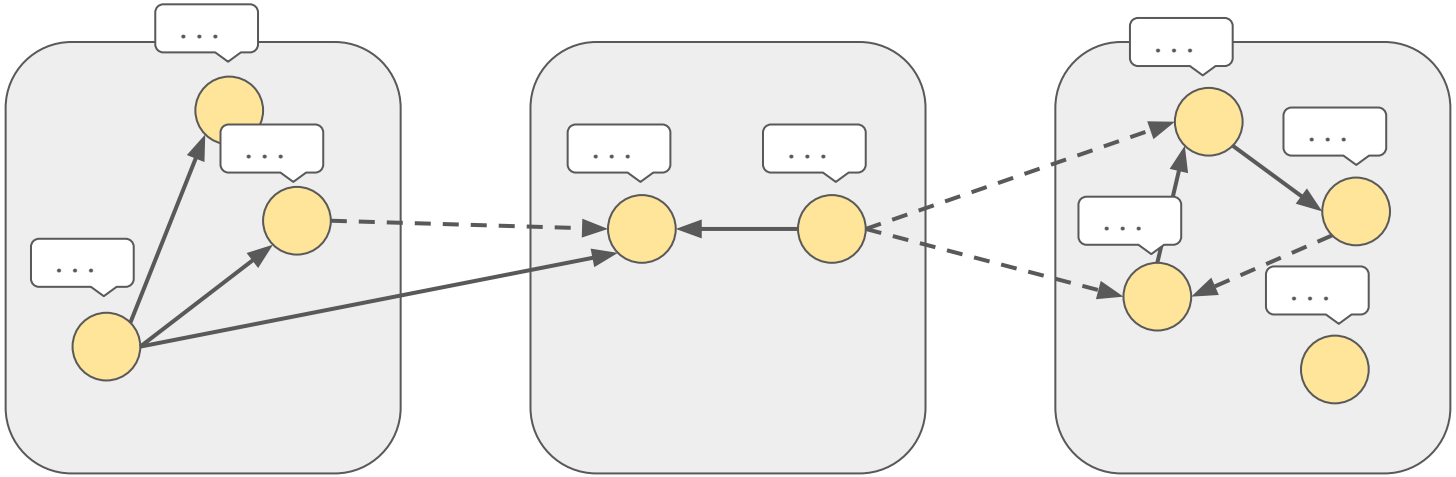

Actor
Reference
m
Message
Knowledge
Aggregator
snapshot(...)

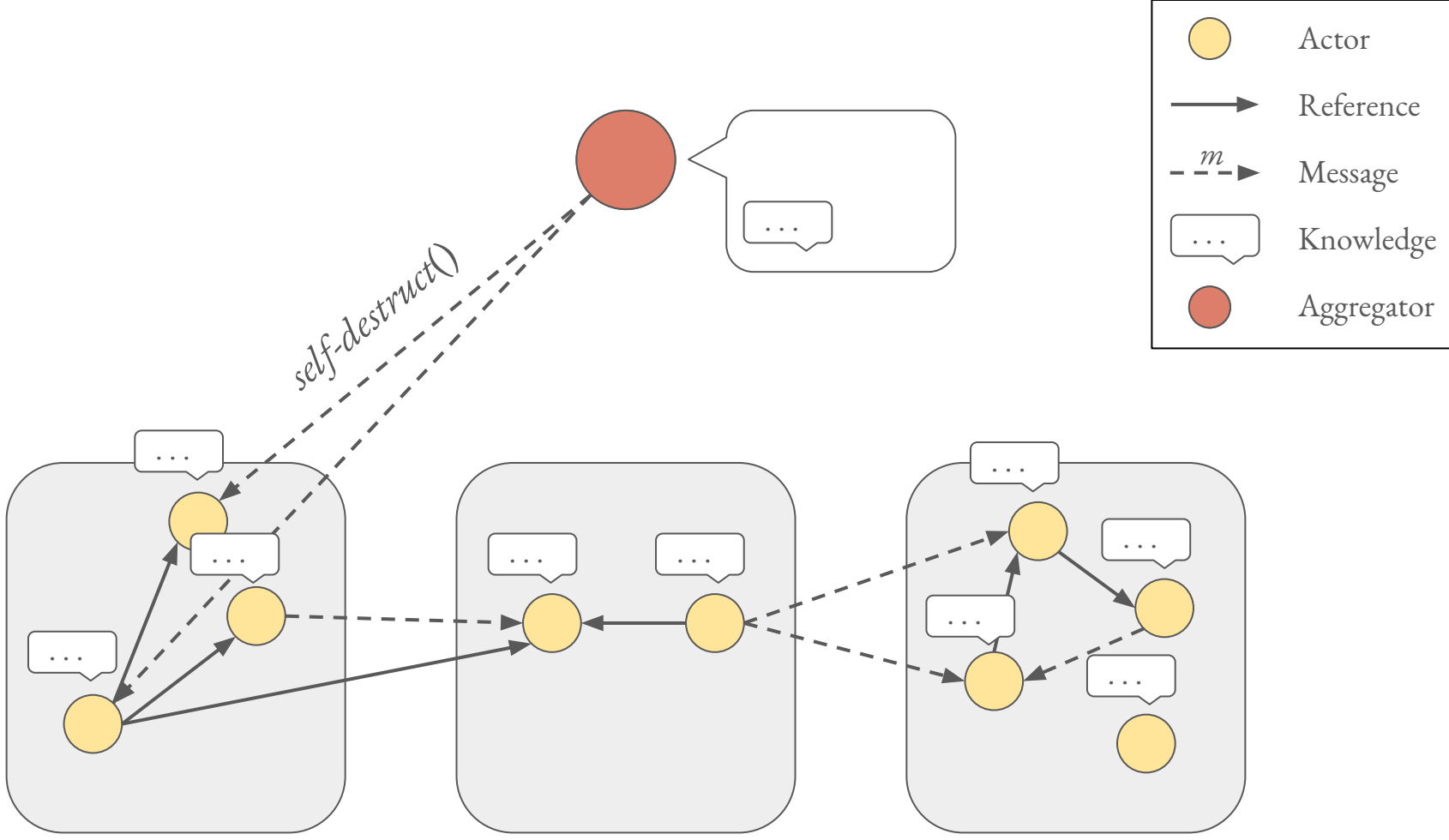

Actor
Reference
m
Message
Knowledge
Aggregator
snapshot(...)
snapshot(...)

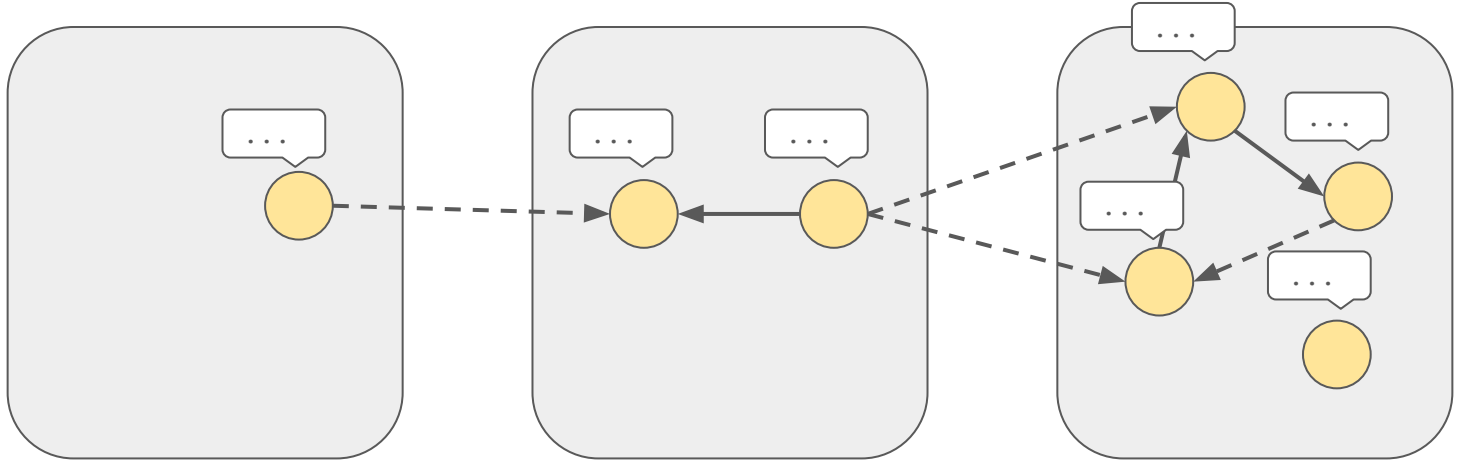

Actor
Reference
m
Message
Knowledge
Aggregator

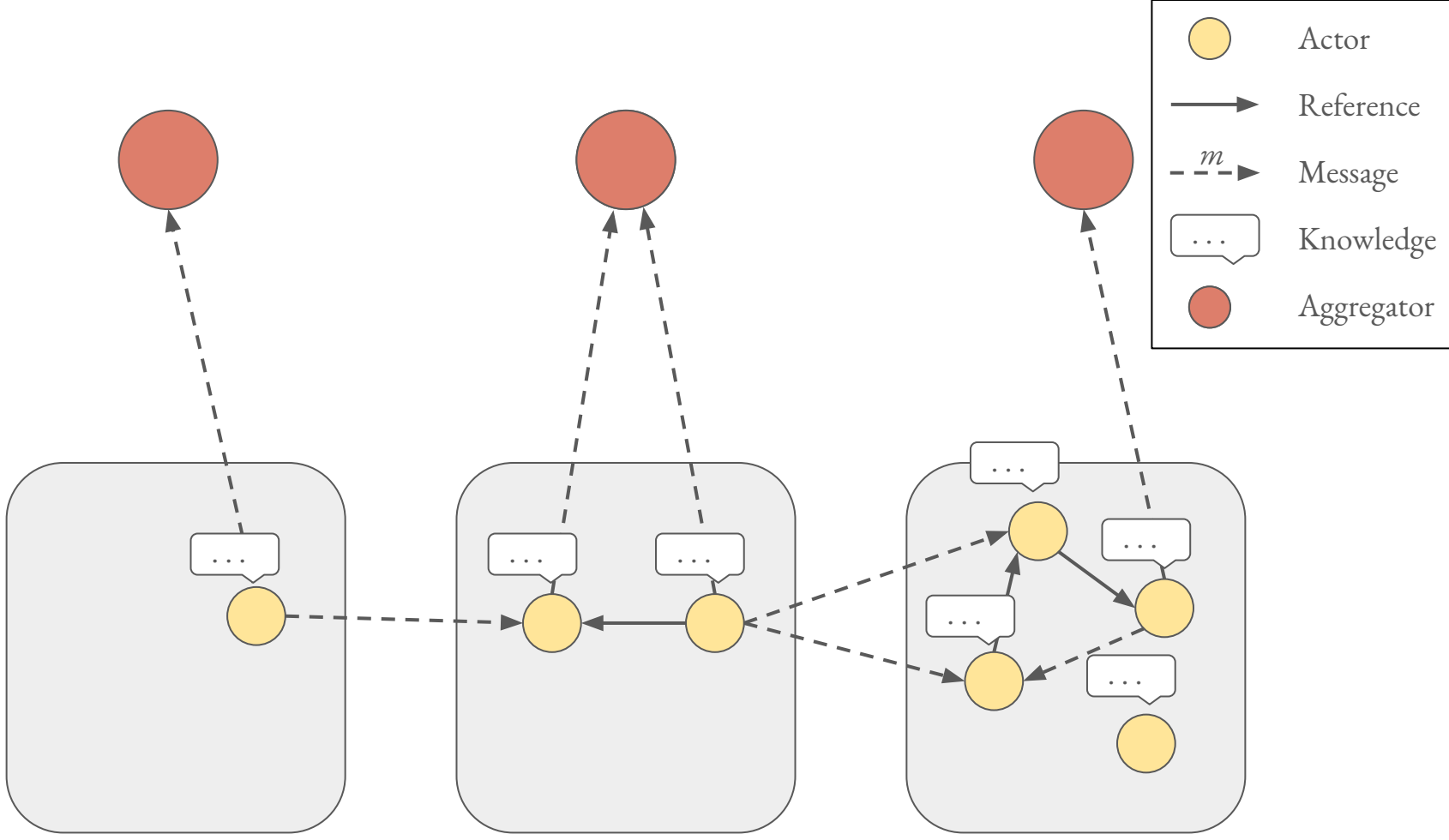


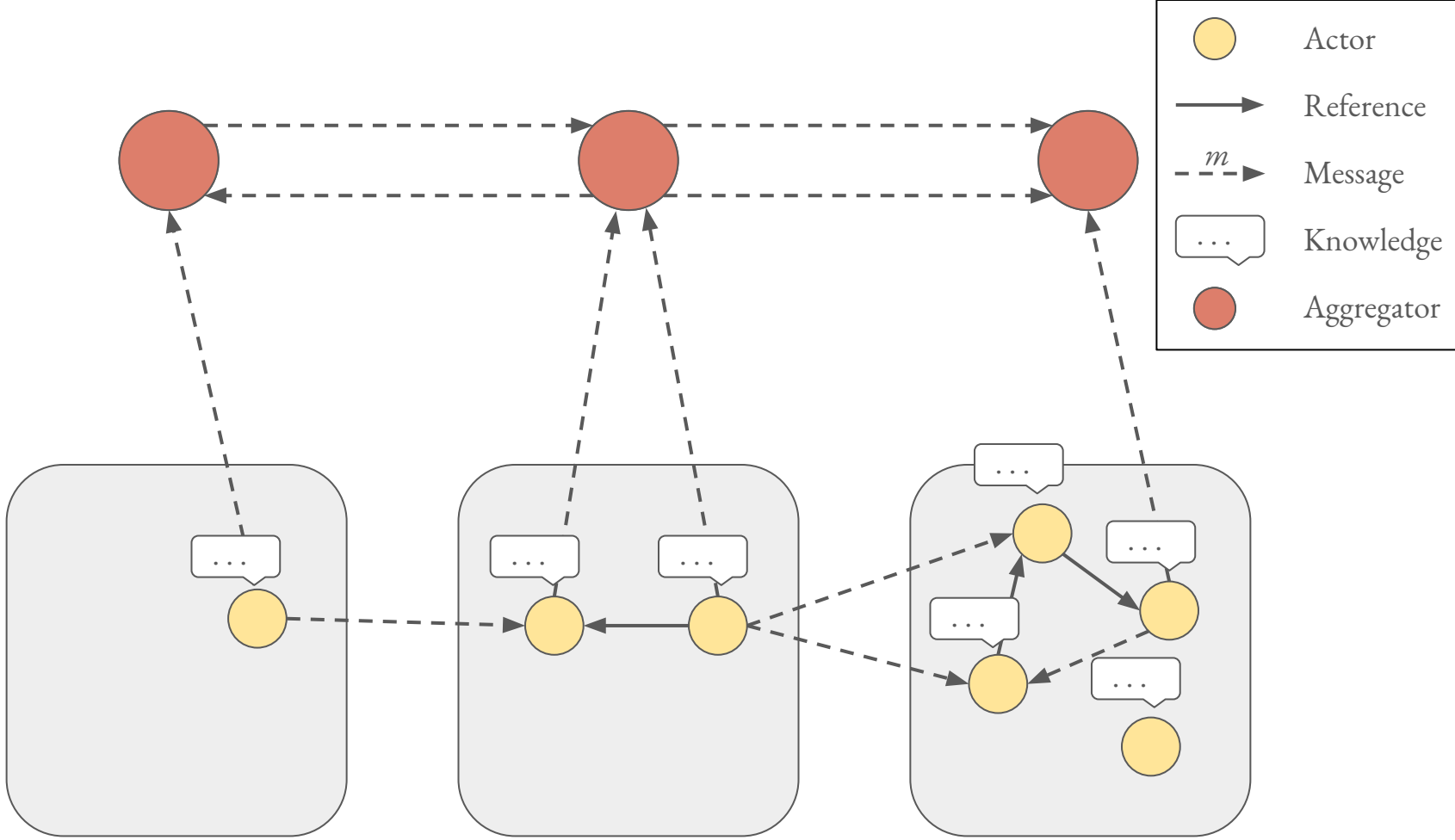

Actor
Reference
m
Message
Knowledge
Aggregator

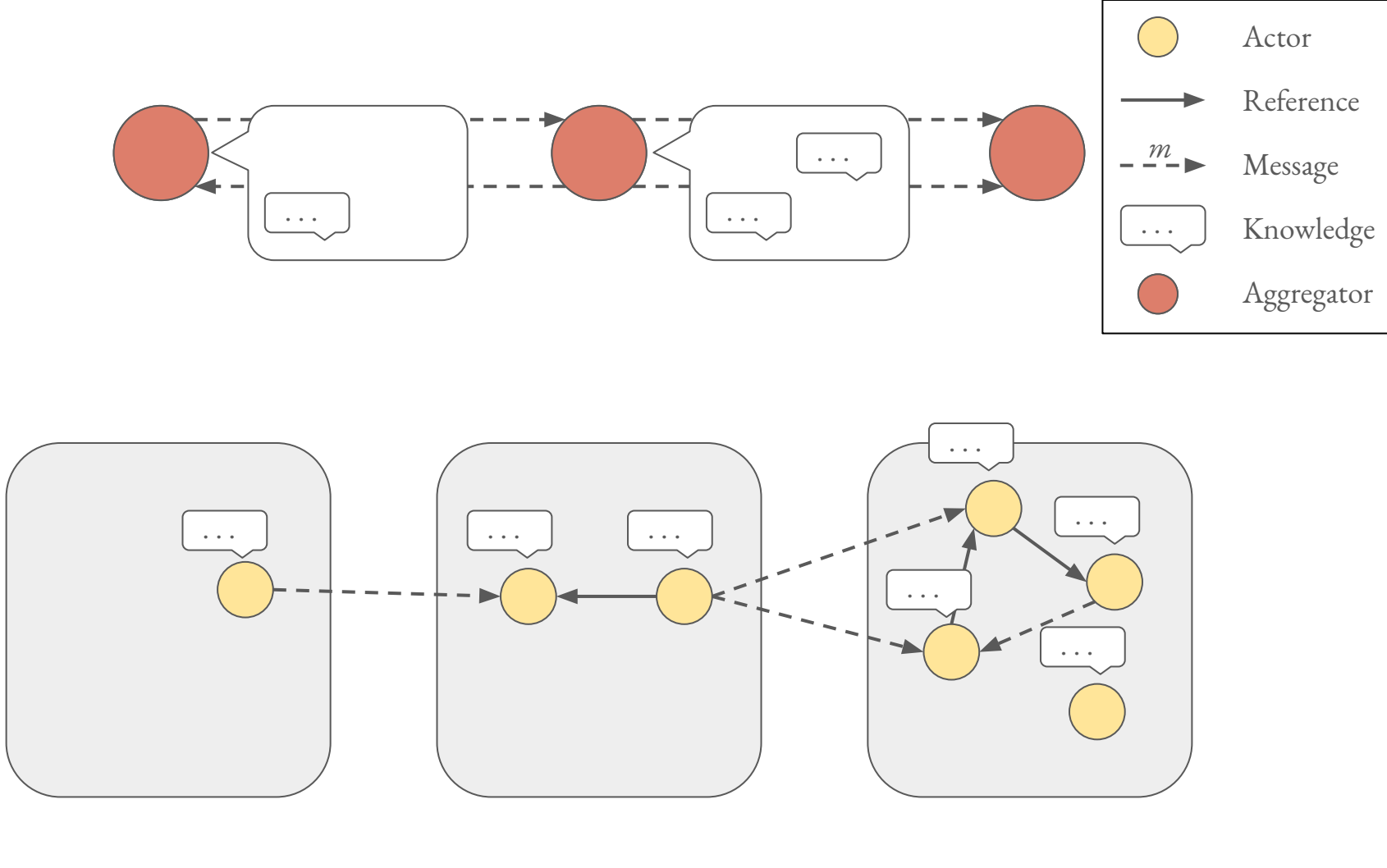


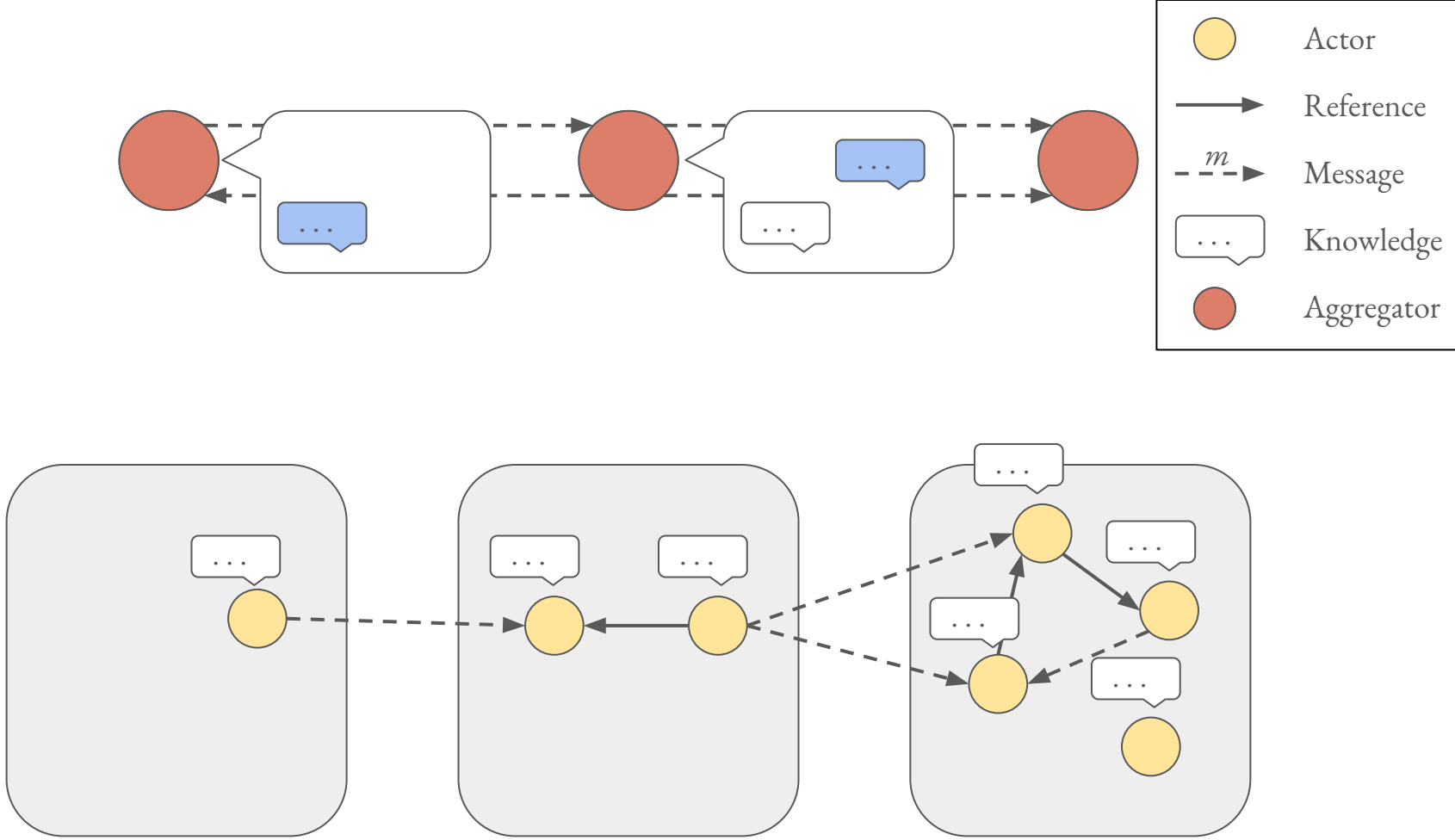

Actor
Reference
m
Message
Knowledge
Aggregator
self-destruct()

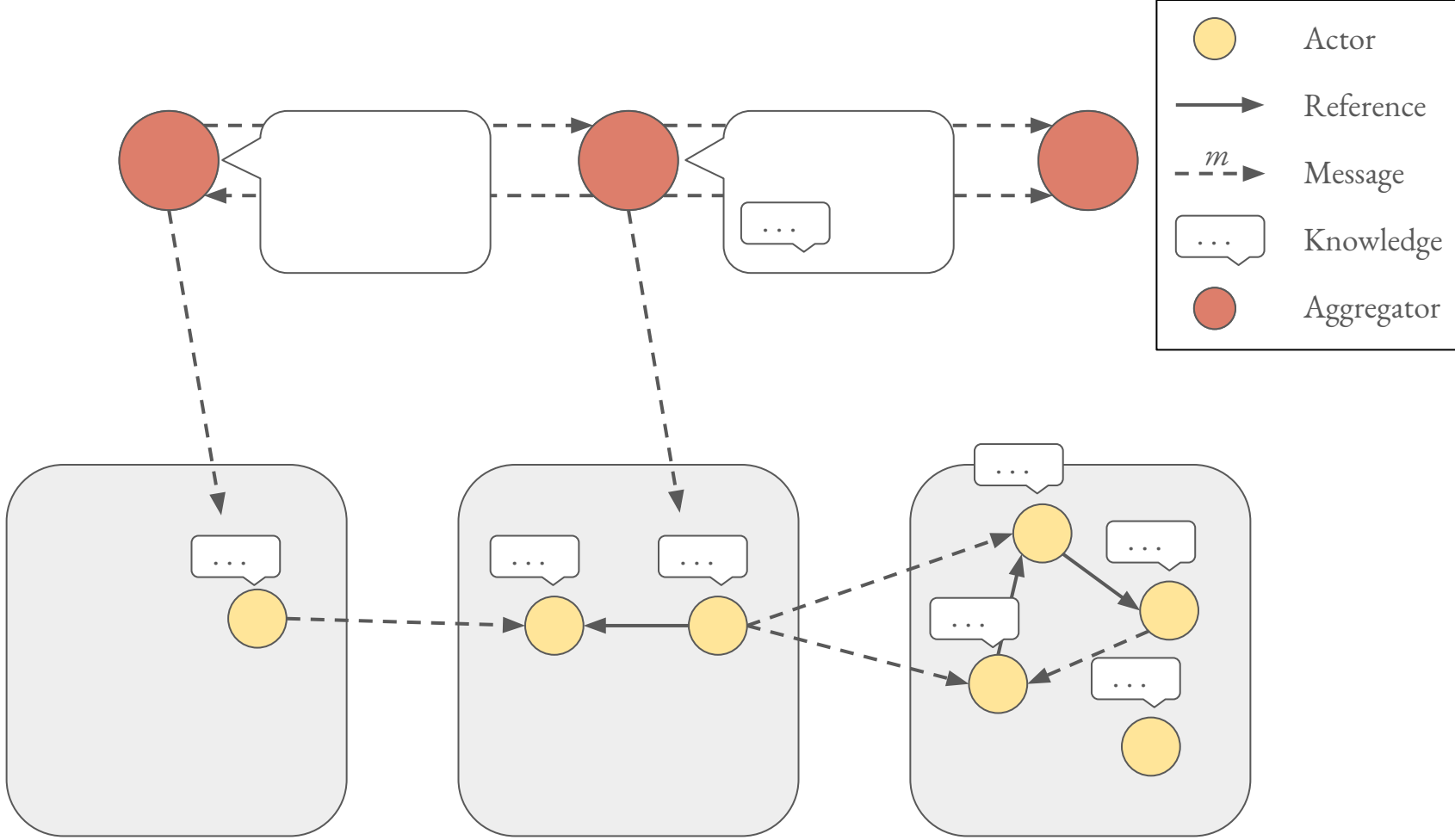

Actor
Reference
m
Message
Knowledge
Aggregator

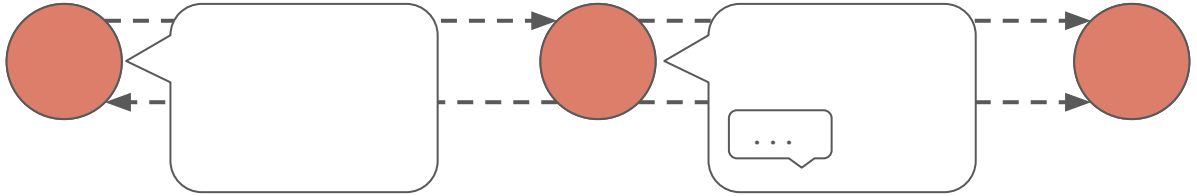

Actor
Reference
m
Message
Knowledge
Aggregator

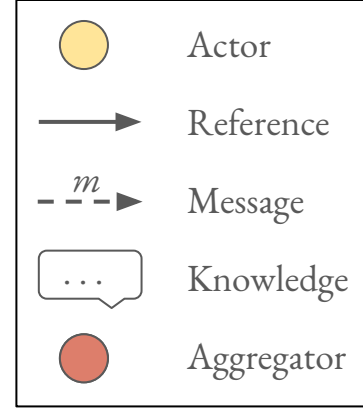

Actor
Reference
m
Message
Knowledge
Aggregator

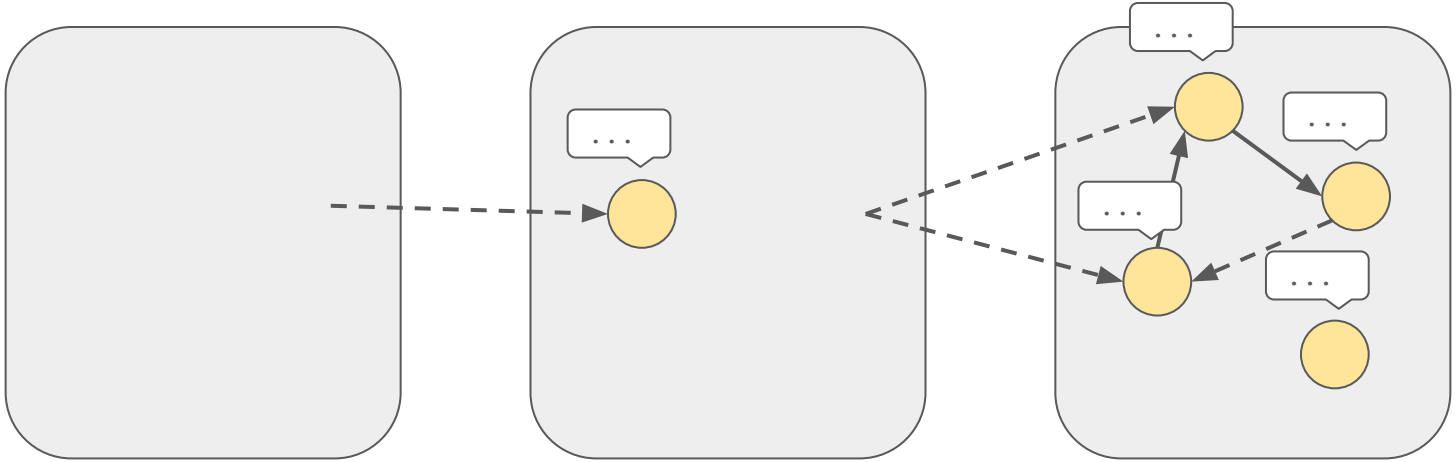

Actor
Reference
m
Message
Knowledge
Aggregator


Actor
Reference
m
Message
Knowledge
Aggregator


Actor
Reference
m
Message
Knowledge
Aggregator


...


Actor
Reference
m
Message
...
Knowledge
Aggregator


...
...
...
...
...

Detecting Terminated Actors

Detecting Terminated Actors

Q: How do you find an actor's **potential inverse acquaintances**?

Q: How do you know if an actor has **undelivered messages**?

Q: How do you know the snapshots are **consistent**?

Detecting Terminated Actors

Q: How do you find an actor's **potential inverse acquaintances**?

With contact tracing!

Q: How do you know if an actor has **undelivered messages**?

Q: How do you know the snapshots are **consistent**?

Detecting Terminated Actors

Q: How do you find an actor's **potential inverse acquaintances**?

With contact tracing!

Q: How do you know if an actor has **undelivered messages**?

With message counts!

Q: How do you know the snapshots are **consistent**?

Detecting Terminated Actors

Q: How do you find an actor's **potential inverse acquaintances**?

With contact tracing!

Q: How do you know if an actor has **undelivered messages**?

With message counts!

Q: How do you know the snapshots are **consistent**?

Magic!

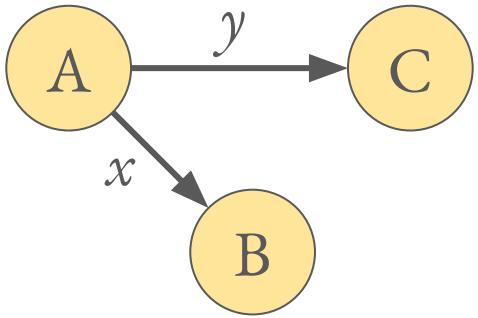
Part III: Contact Tracing

Contact Tracing

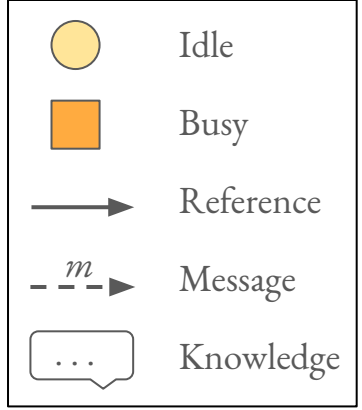
- Actors must use *reference objects* (**refobs**) instead of ordinary references
- Refobs are denoted (x : A ⁻◦ B), where x is a globally unique **token**
- Can only be used by the **owner** A to send messages to the **target** B
- Must be **deactivated** when no longer needed
- Actor gets a refob when it spawns a child
- If A has (x : A ⁻◦ B) and (y : A ⁻◦ C) then A can create (z : C ⁻◦ B)

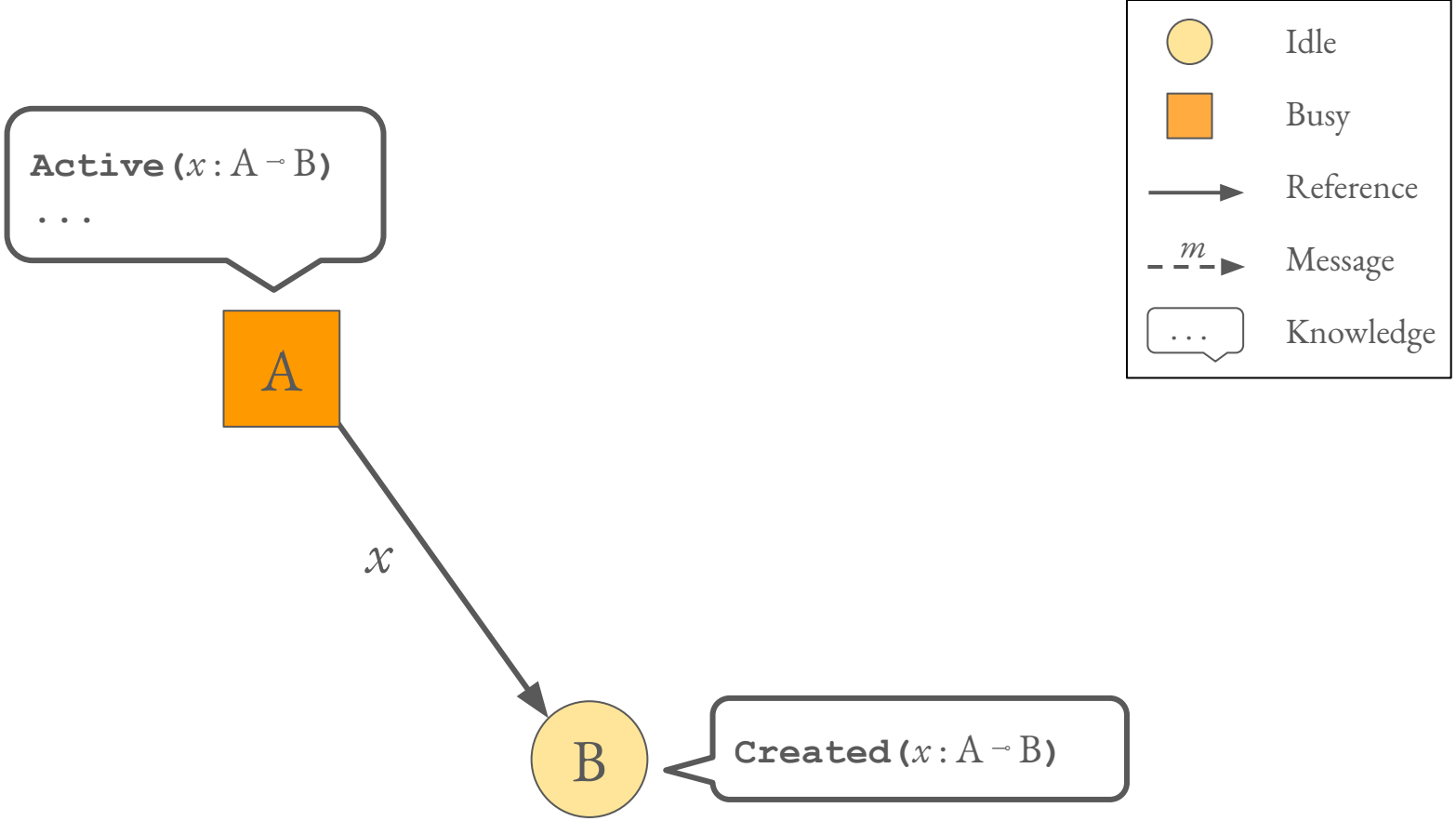
Contact Tracing

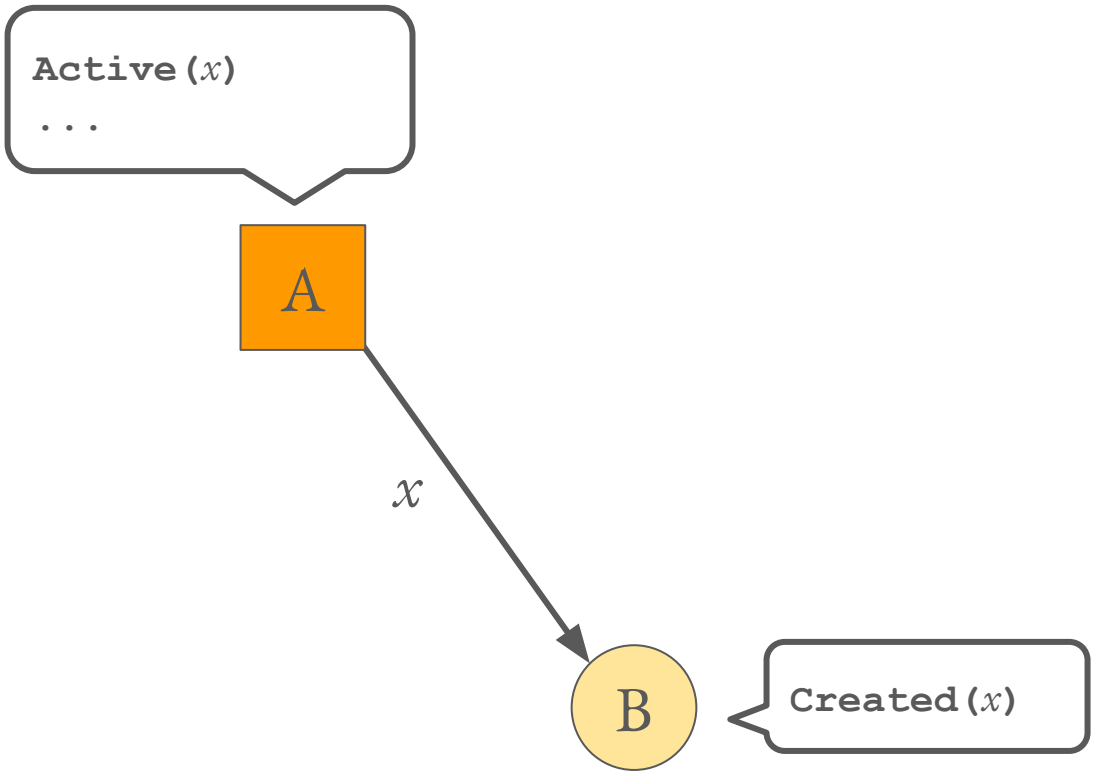
- Actors must use *reference objects* (**refobs**) instead of ordinary references
- Refobs are denoted ($x : \mathrm{A} \multimap \mathrm{B}$), where x is a globally unique **token**
- Can only be used by the **owner** A to send messages to the **target** B
- Must be **deactivated** when no longer needed
- Actor gets a refob when it spawns a child
- If A has ($x : \mathrm{A} \multimap \mathrm{B}$) and ($y : \mathrm{A} \multimap \mathrm{C}$) then A can create ($z : \mathrm{C} \multimap \mathrm{B}$)

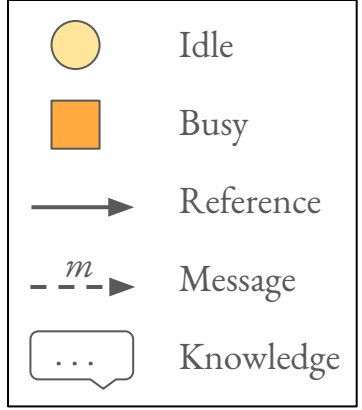


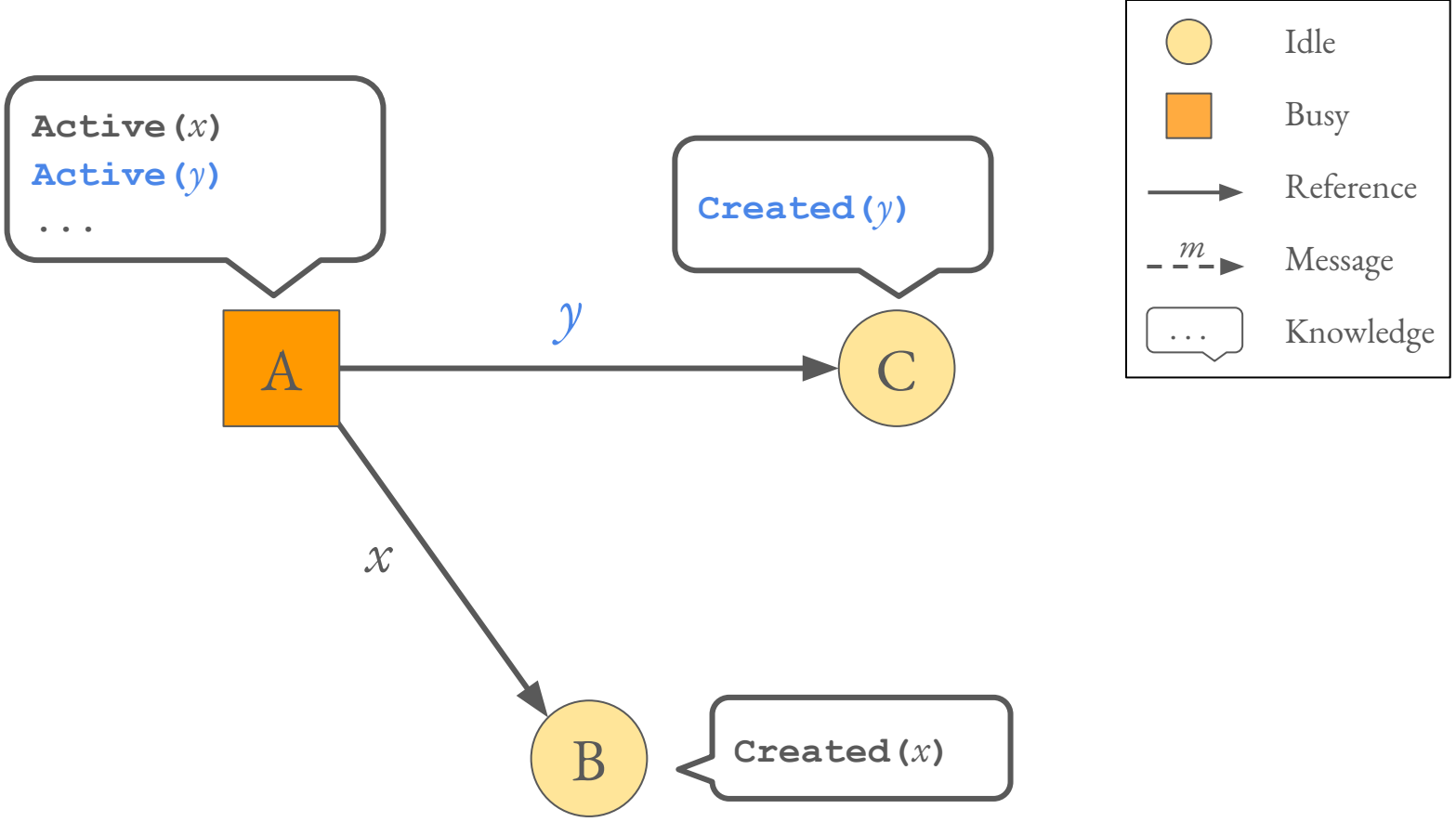

A

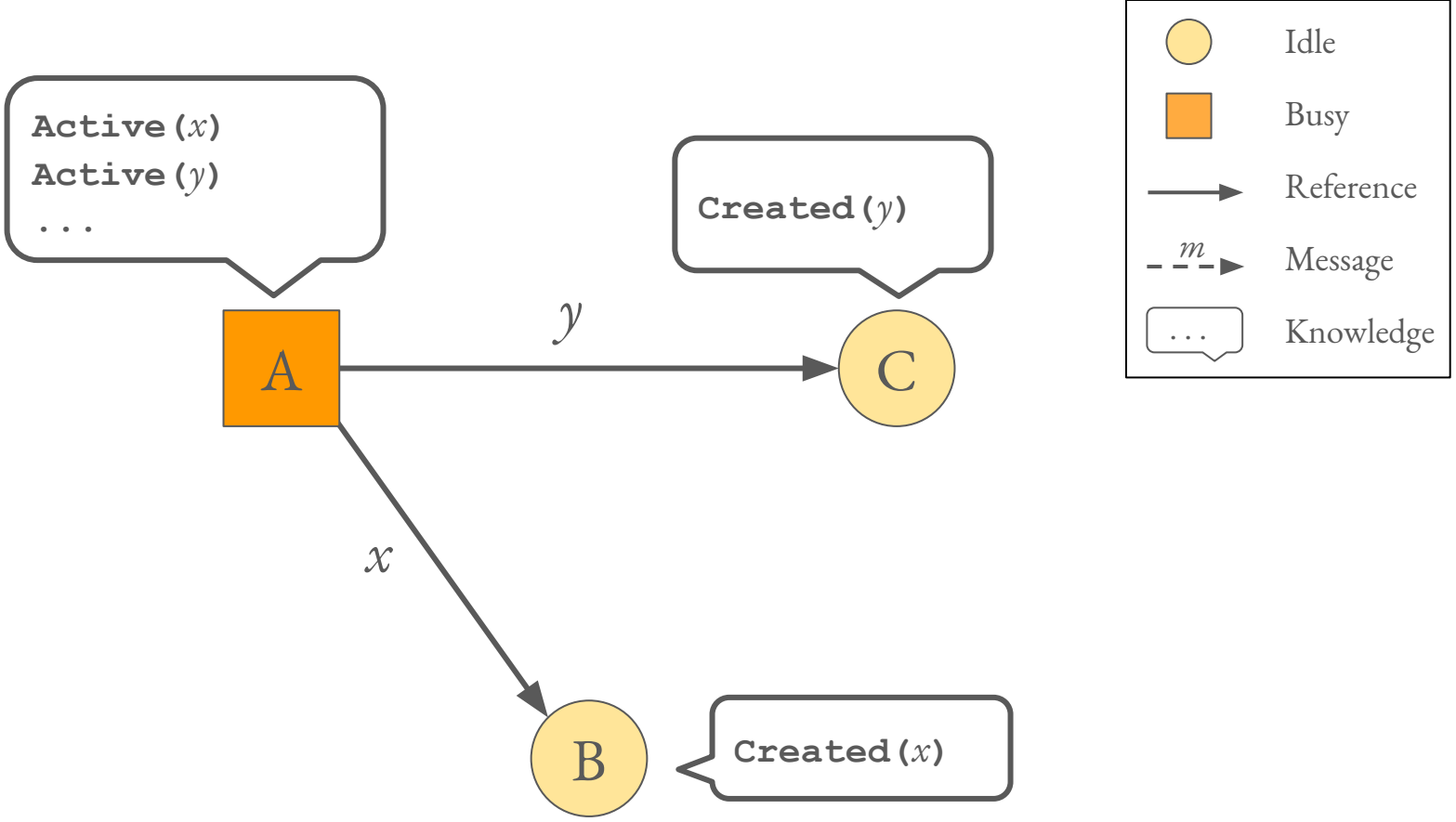

Idle
Busy
Reference
m
Message
Knowledge

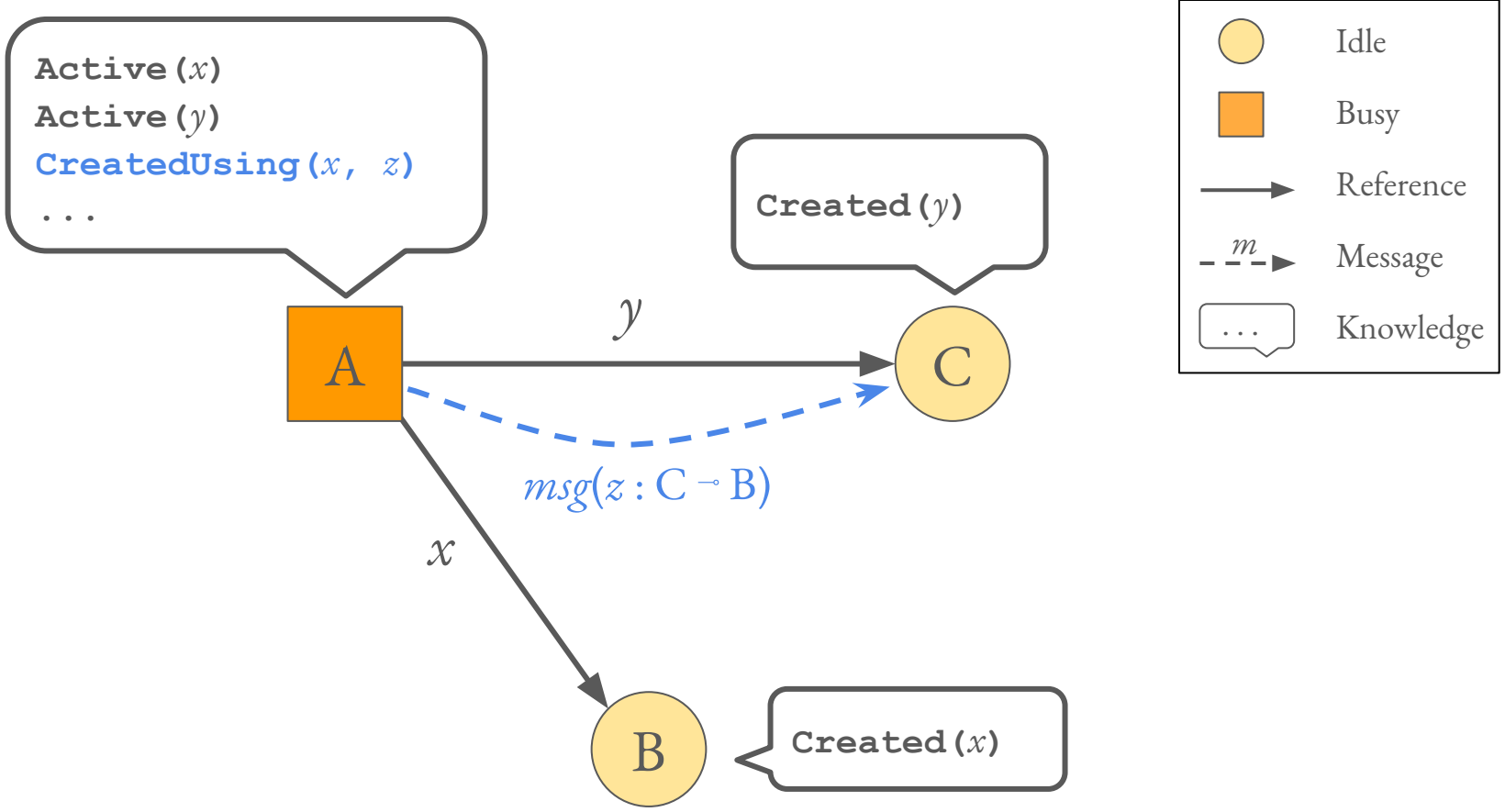

Active(x : A ⊸ B)
. . .
A
x
B
Created(x : A ⊸ B)
Idle
Busy
Reference
m
Message
. . .
Knowledge

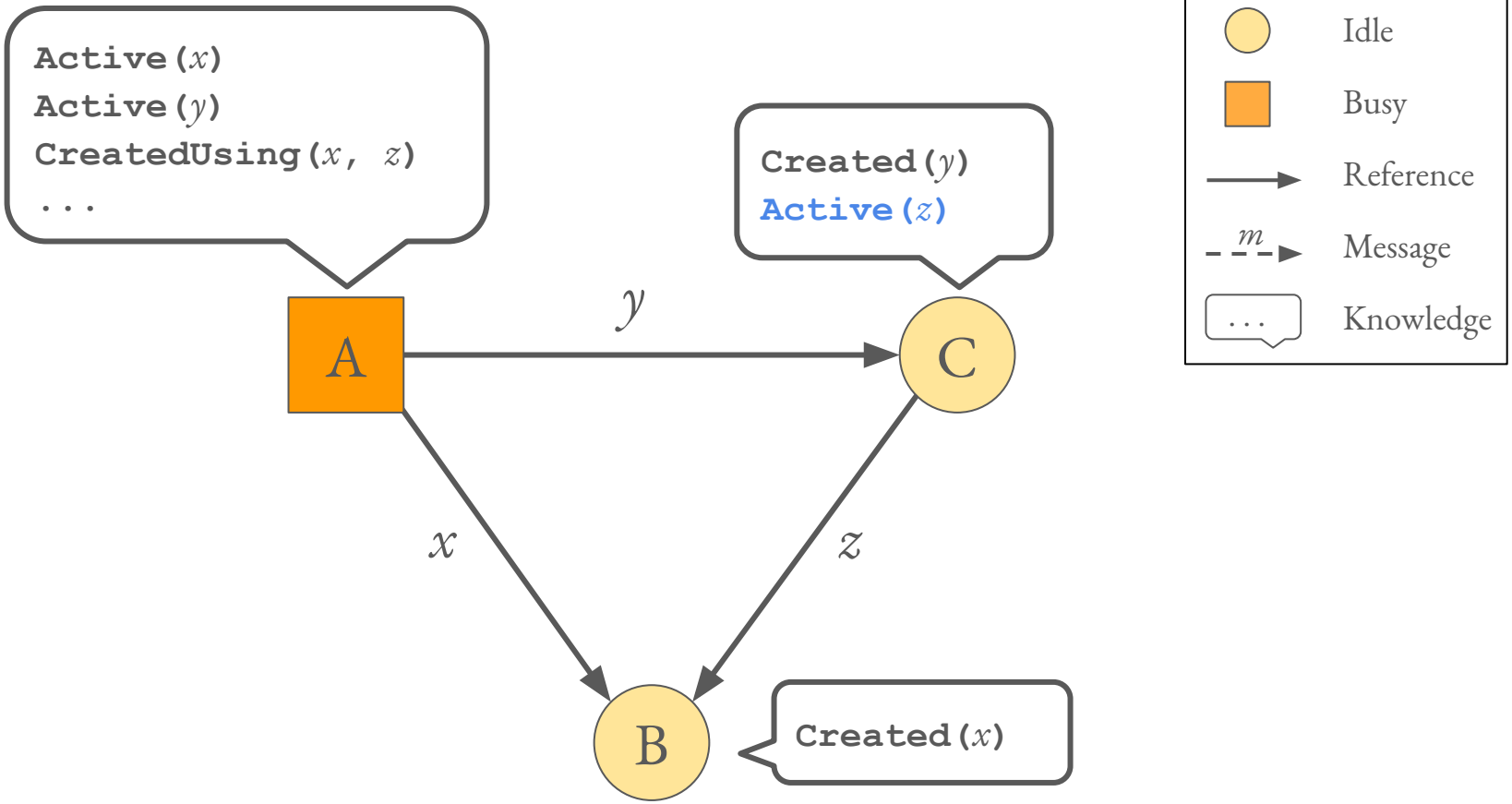

Active(x)
...
A
x
B
Created(x)

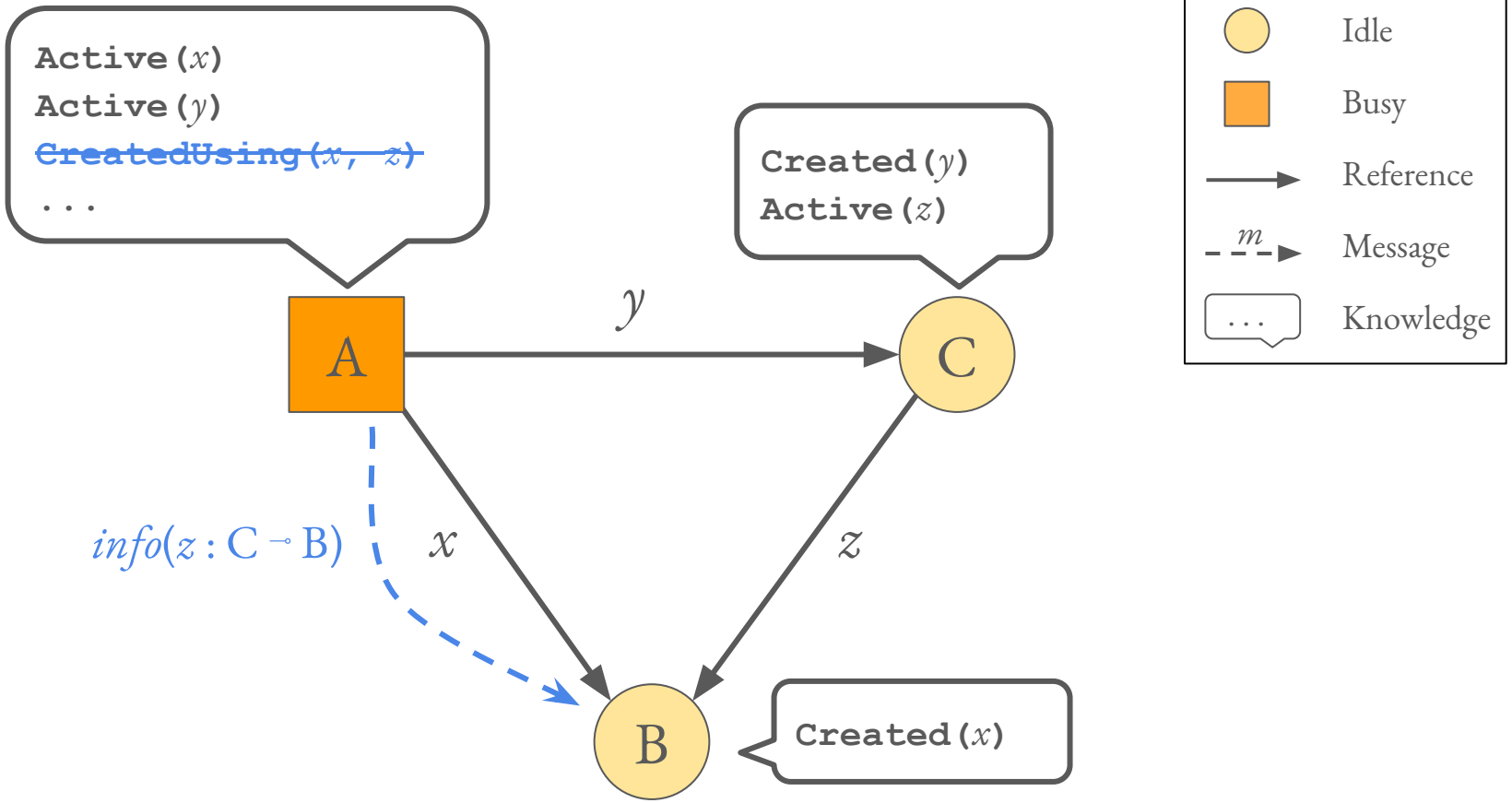

Idle
Busy
Reference
m
Message
...
Knowledge

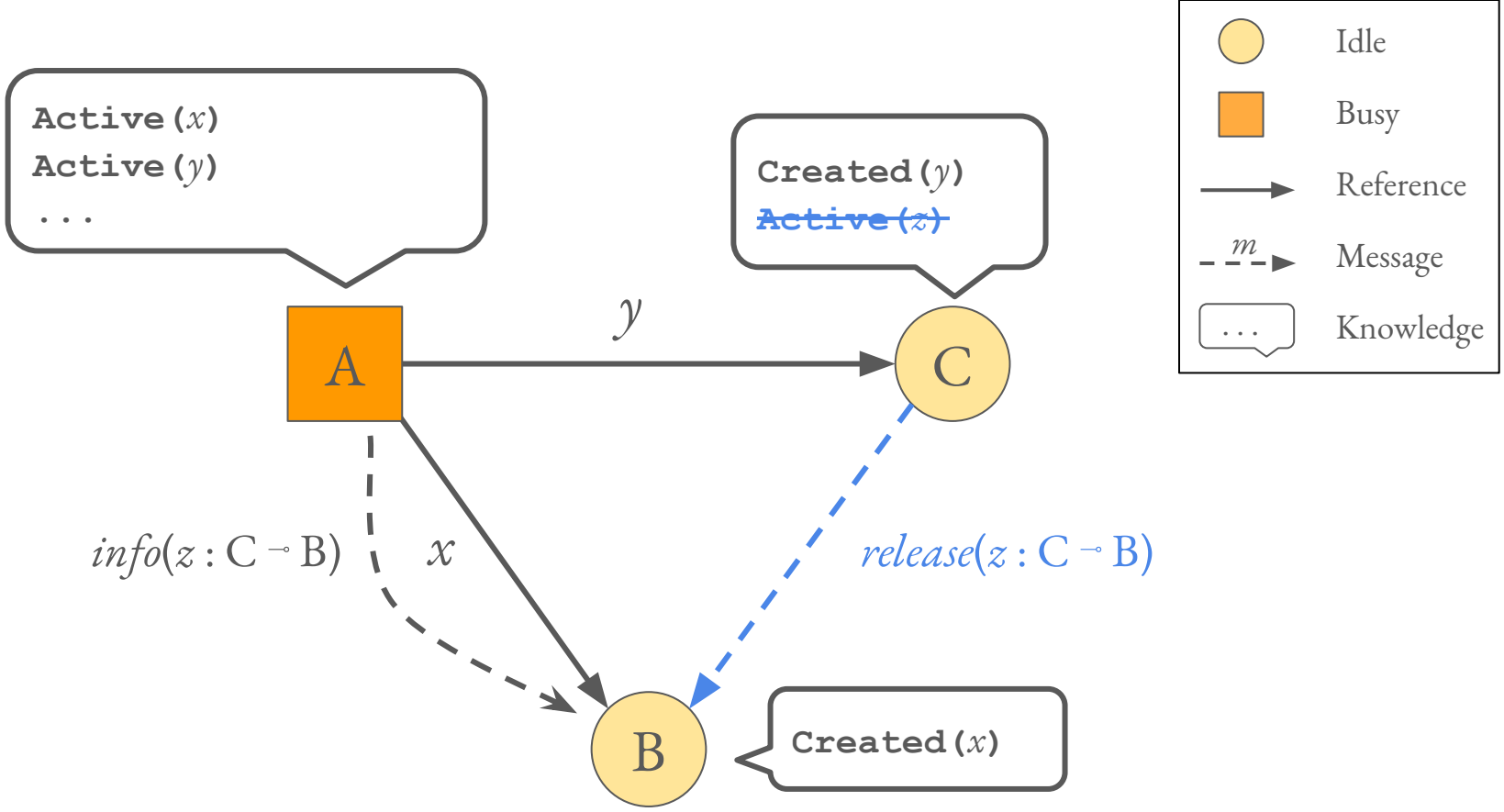

Active(x)
Active(y)
. . .
Created(y)
A
y
C
x
B
Created(x)
Idle
Busy
Reference
m
Message
. . .
Knowledge


Active(x)
Active(y)
...
Created(y)
A
y
C
x
B
Created(x)
Idle
Busy
Reference
m
Message
...
Knowledge

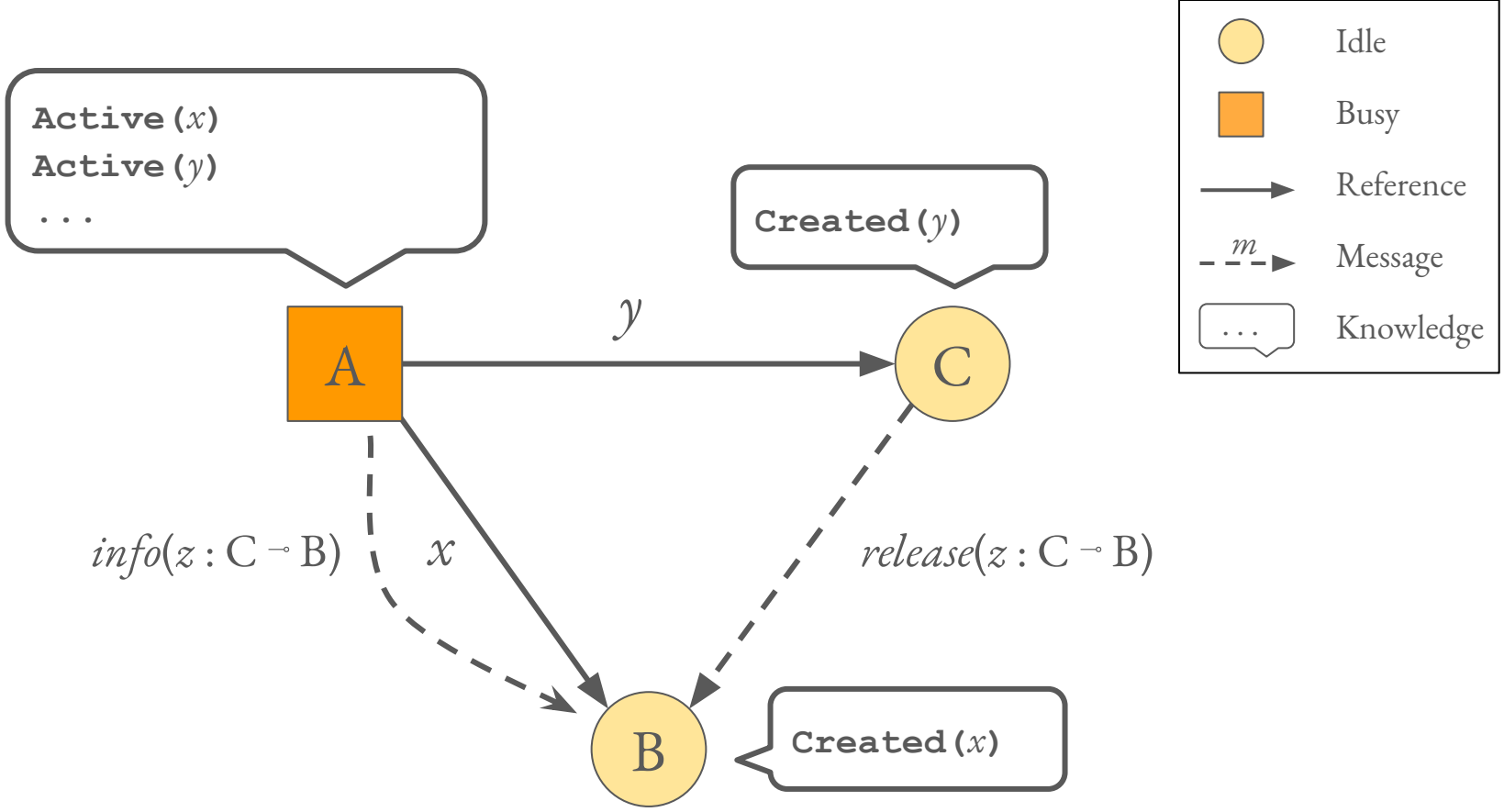

Active(x)
Active(y)
CreatedUsing(x, z)
...
Created(y)
A
y
C
msg(z : C → B)
x
B
Created(x)
Idle
Busy
Reference
m
Message
...
Knowledge

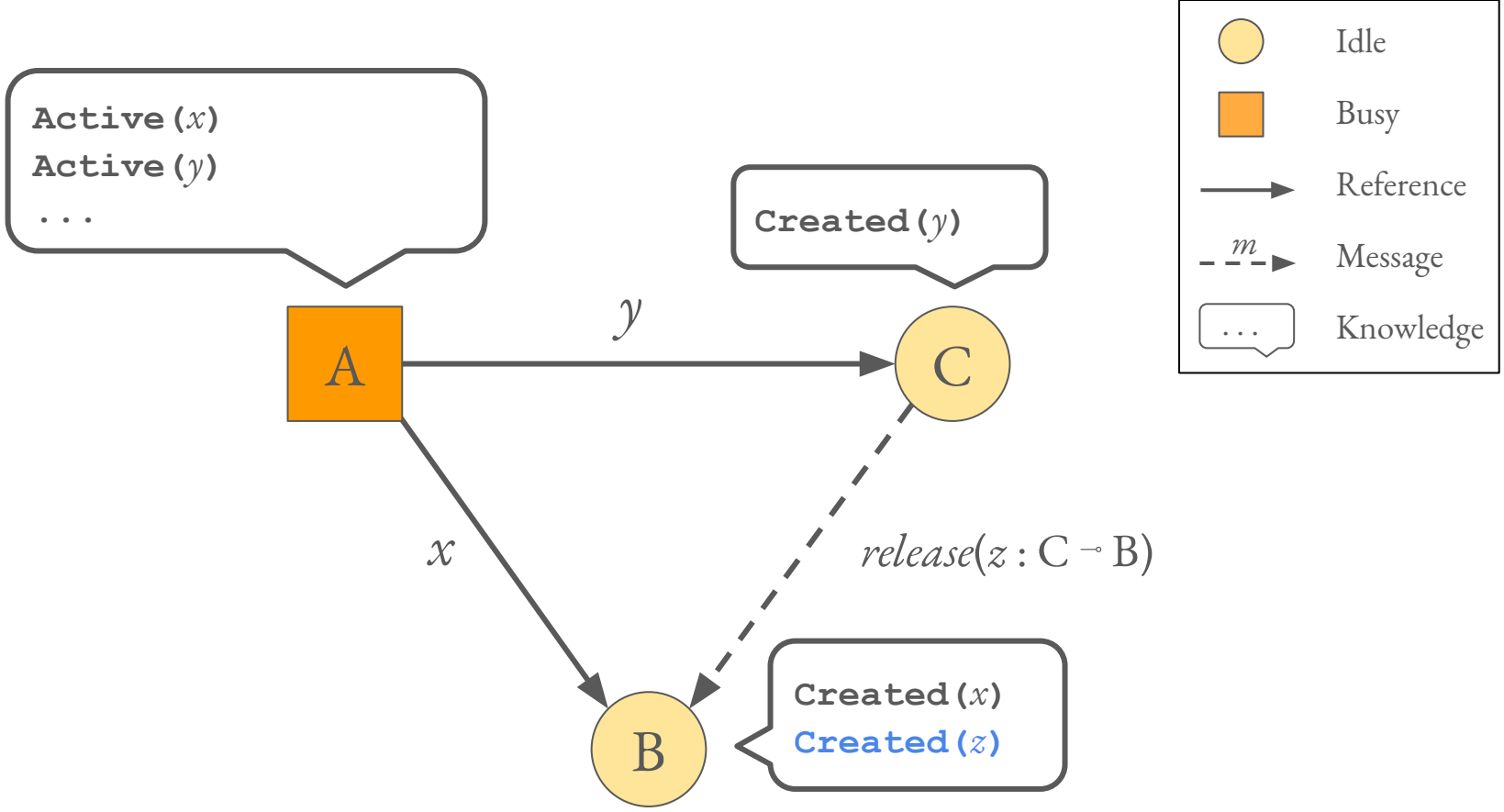

Active(x)
Active(y)
CreatedUsing(x, z)
...
Created(y)
Active(z)
A
C
B
y
x
z
Created(x)
Idle
Busy
Reference
m
Message
Knowledge

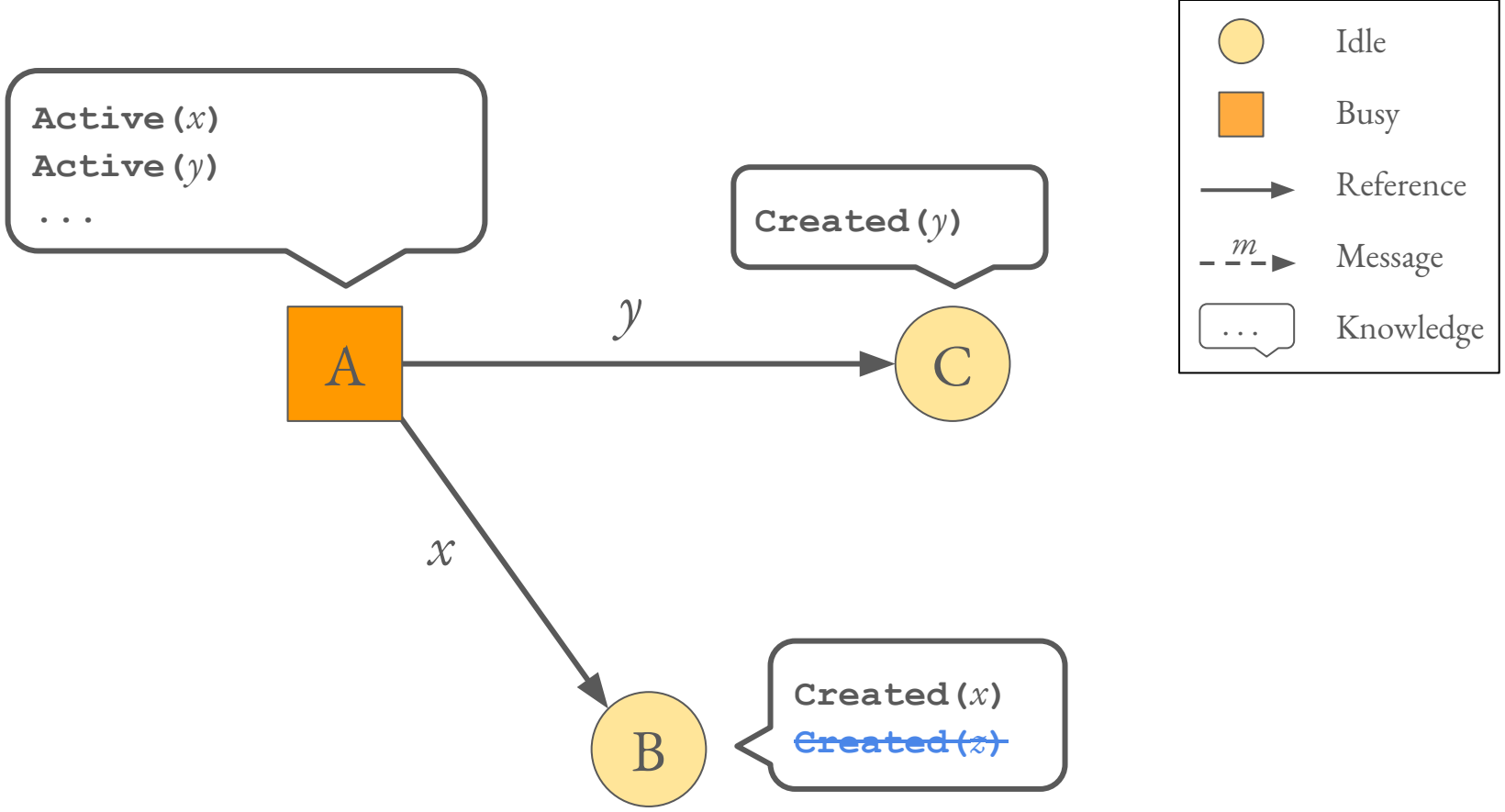

Active(x)
Active(y)
CreatedUsing(x, z)
...
Created(y)
Active(z)
A
C
B
y
x
z
info(z : C → B)
Created(x)
Idle
Busy
Reference
m
Message
Knowledge


Active(x)
Active(y)
...
Created(y)
Active(z)
A
y
C
info(z : C ⊸ B)
x
release(z : C ⊸ B)
B
Created(x)
Idle
Busy
Reference
m
Message
...
Knowledge

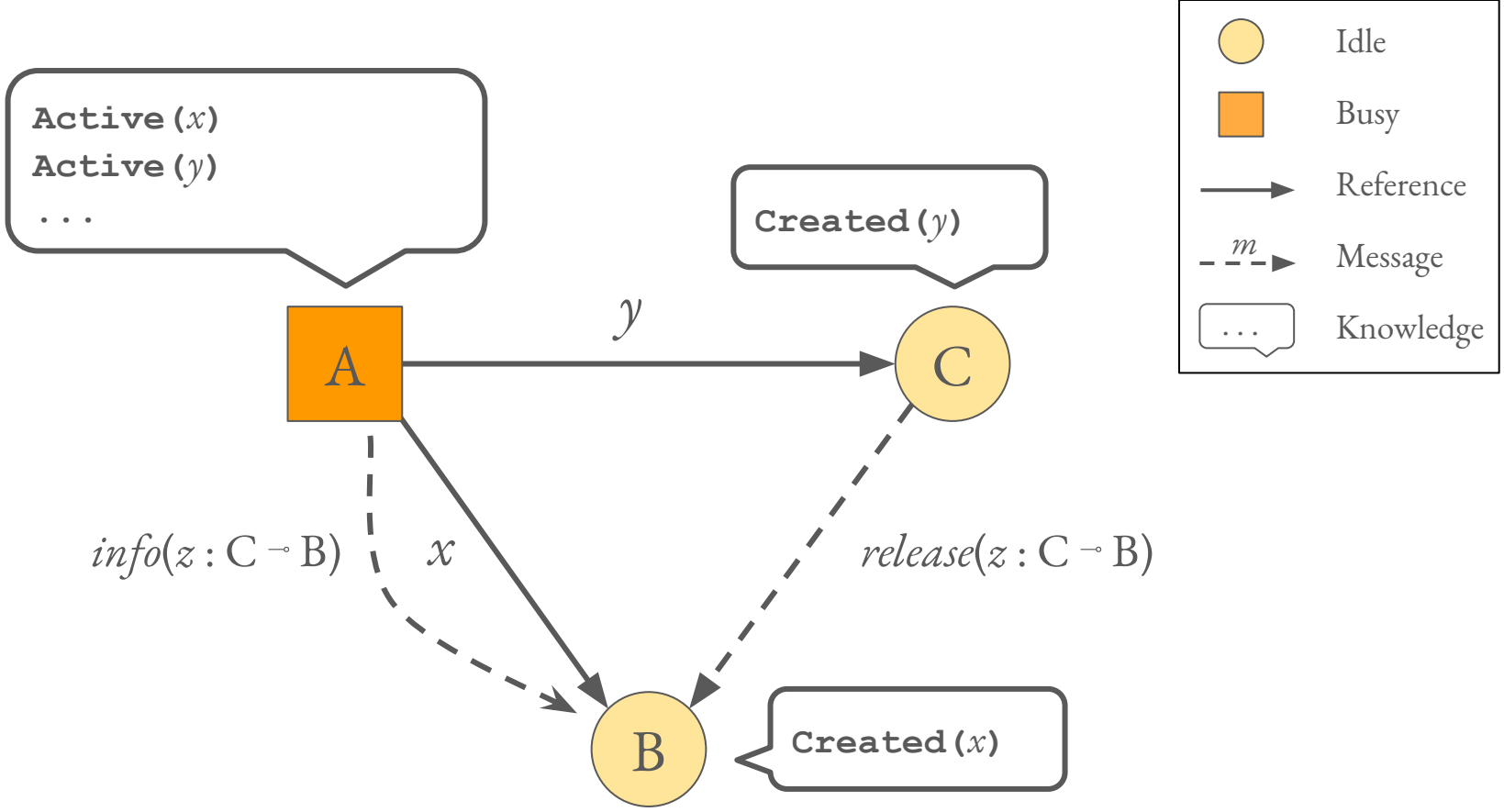
Case 1: *info* message arrives first

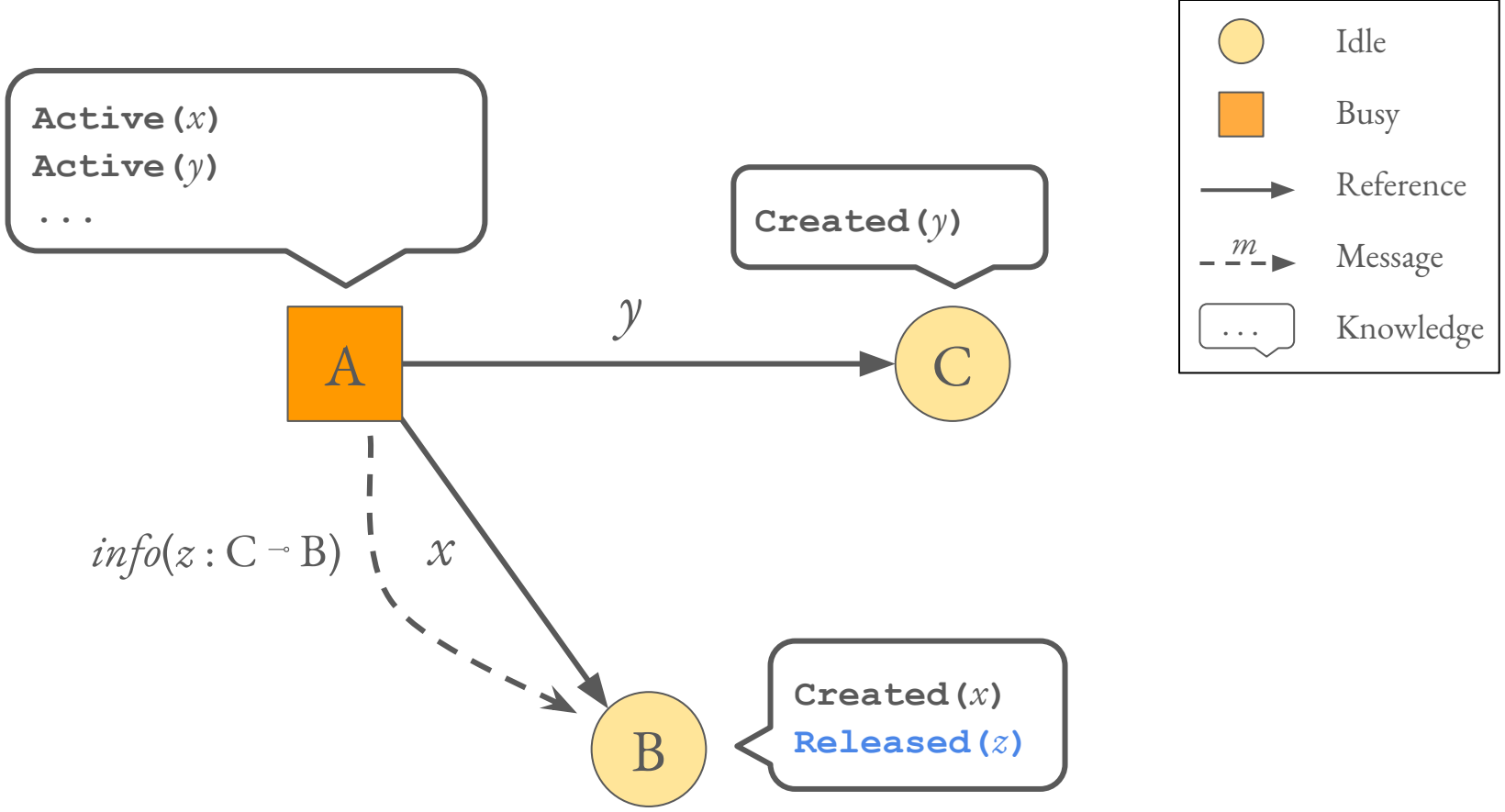

Active(x)
Active(y)
...
Created(y)
A
y
C
info(z : C ⊸ B)
x
release(z : C ⊸ B)
B
Created(x)
Idle
Busy
Reference
m
Message
...
Knowledge

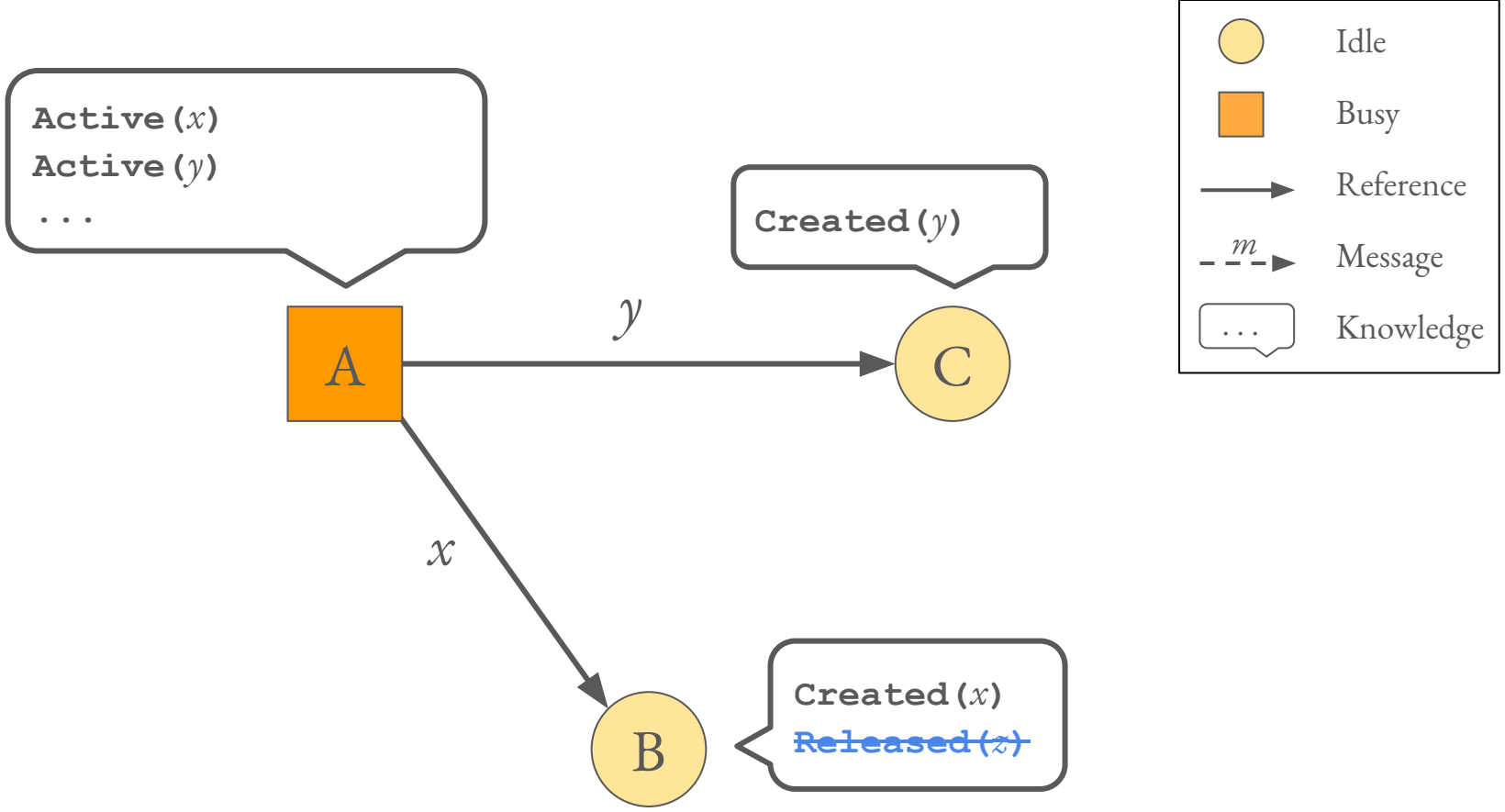

Active(x)
Active(y)
...
Created(y)
A
y
C
x
release(z : C ⊸ B)
B
Created(x)
Created(z)
Idle
Busy
Reference
m
Message
...
Knowledge


Active(x)
Active(y)
...
Created(y)
A
y
C
x
B
Created(x)
Created(z)
Idle
Busy
Reference
m
Message
...
Knowledge

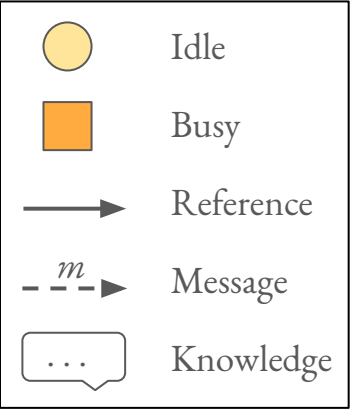
Case 2: *release* message arrives first


Active(x)
Active(y)
...
Created(y)
A
y
C
info(z : C ⊸ B)
x
release(z : C ⊸ B)
B
Created(x)
Idle
Busy
Reference
m
Message
...
Knowledge

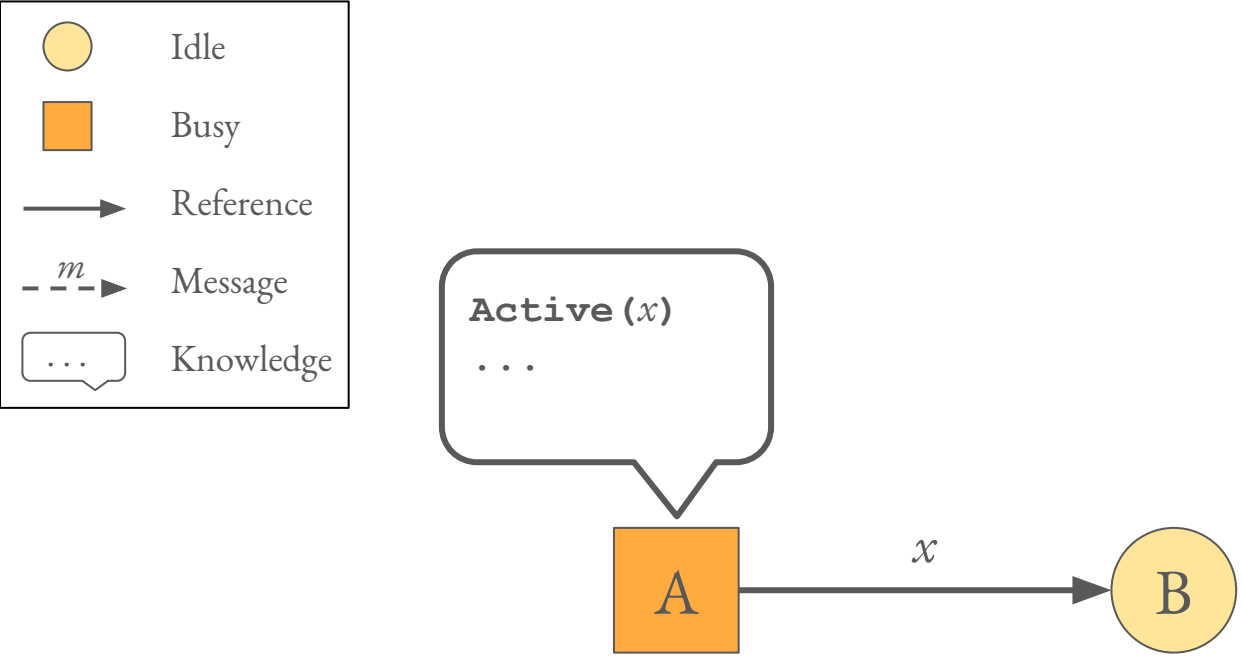

Active(x)
Active(y)
...
Created(y)
A
y
C
info(z : C ⊸ B)
x
B
Created(x)
Released(z)
Idle
Busy
Reference
m
Message
...
Knowledge

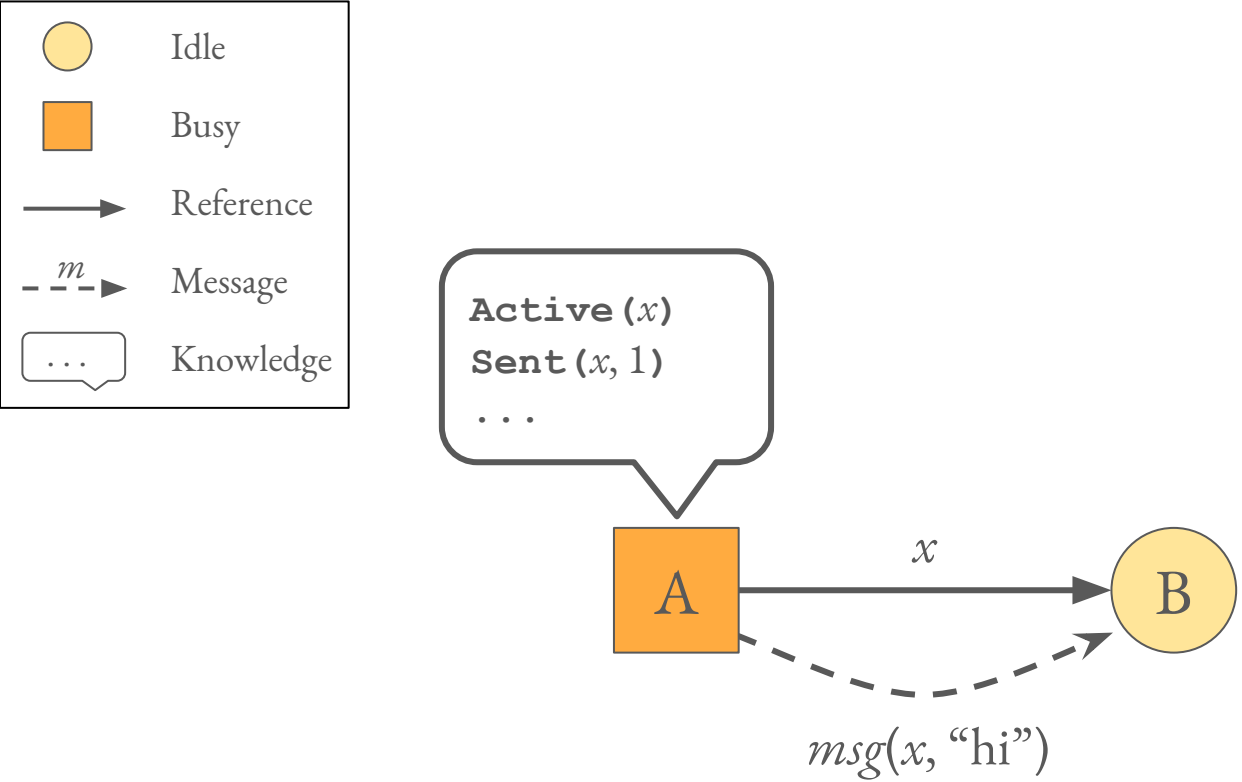

Active(x)
Active(y)
...
Created(y)
A
y
C
x
B
Created(x)
Released(z)
Idle
Busy
Reference
m
Message
...
Knowledge

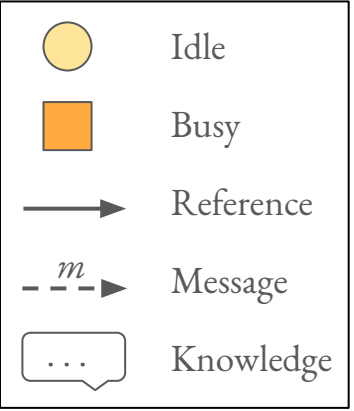
Part IV: Message Counts

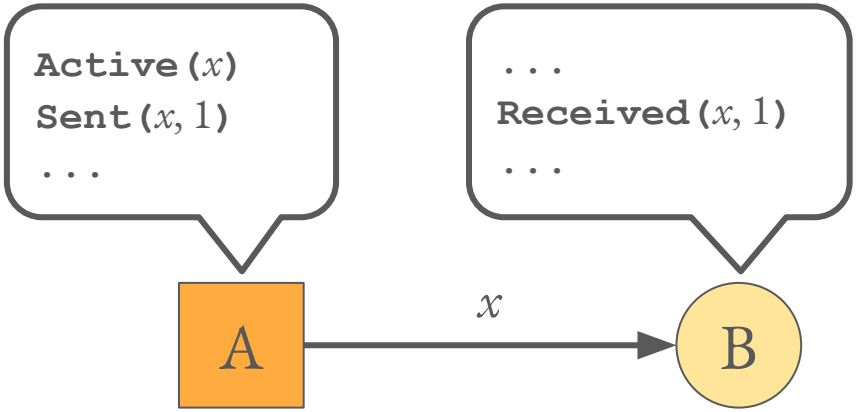

Idle
Busy
Reference
m
Message
Knowledge

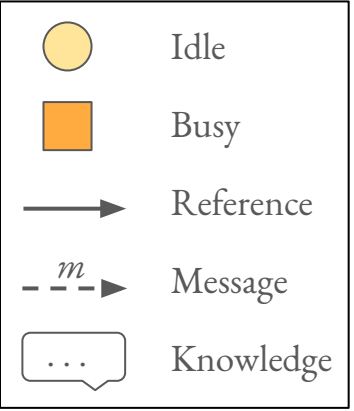

msg(x : A ⊸ B)
A
B

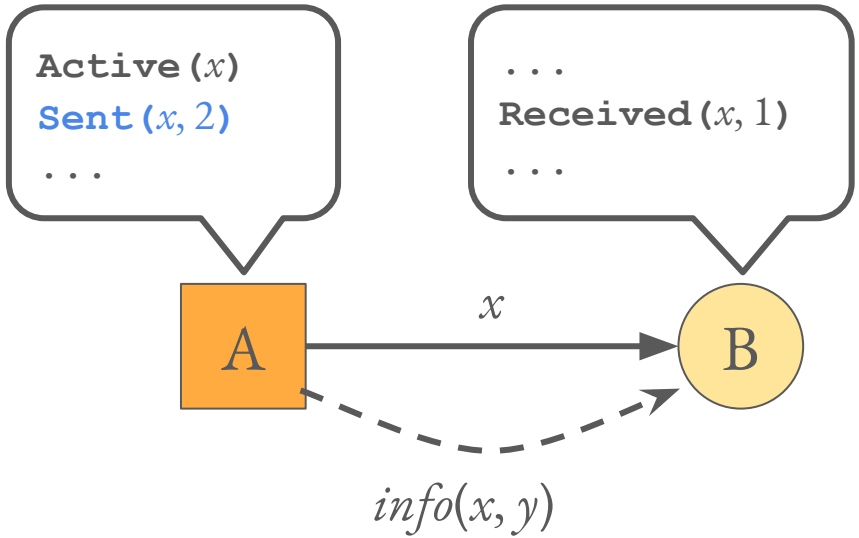

Idle
Busy
Reference
m
Message
Knowledge
Active(x)
. . .
A
x
B

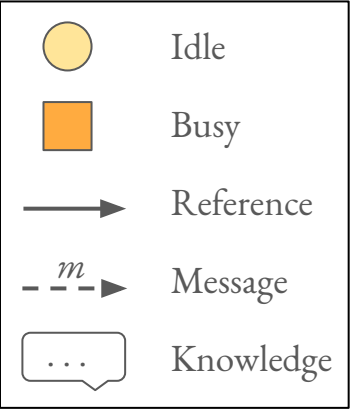

Idle
Busy
Reference
m
Message
Knowledge
Active(x)
Sent(x, 1)
. . .
A
x
B
msg(x, "hi")

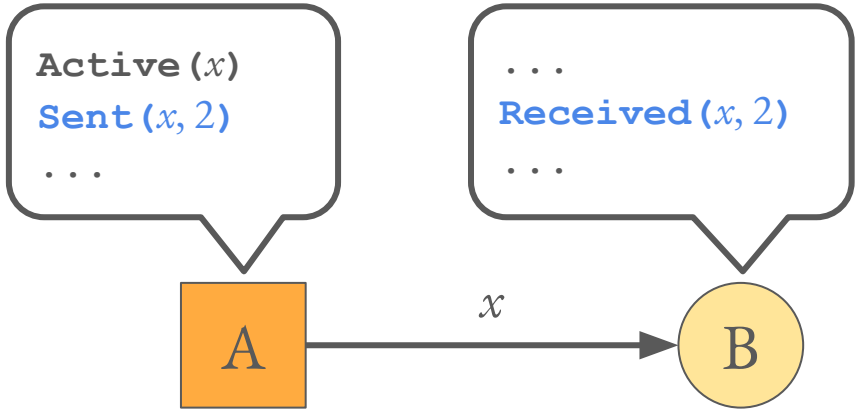

Idle
Busy
Reference
m
Message
. . .
Knowledge

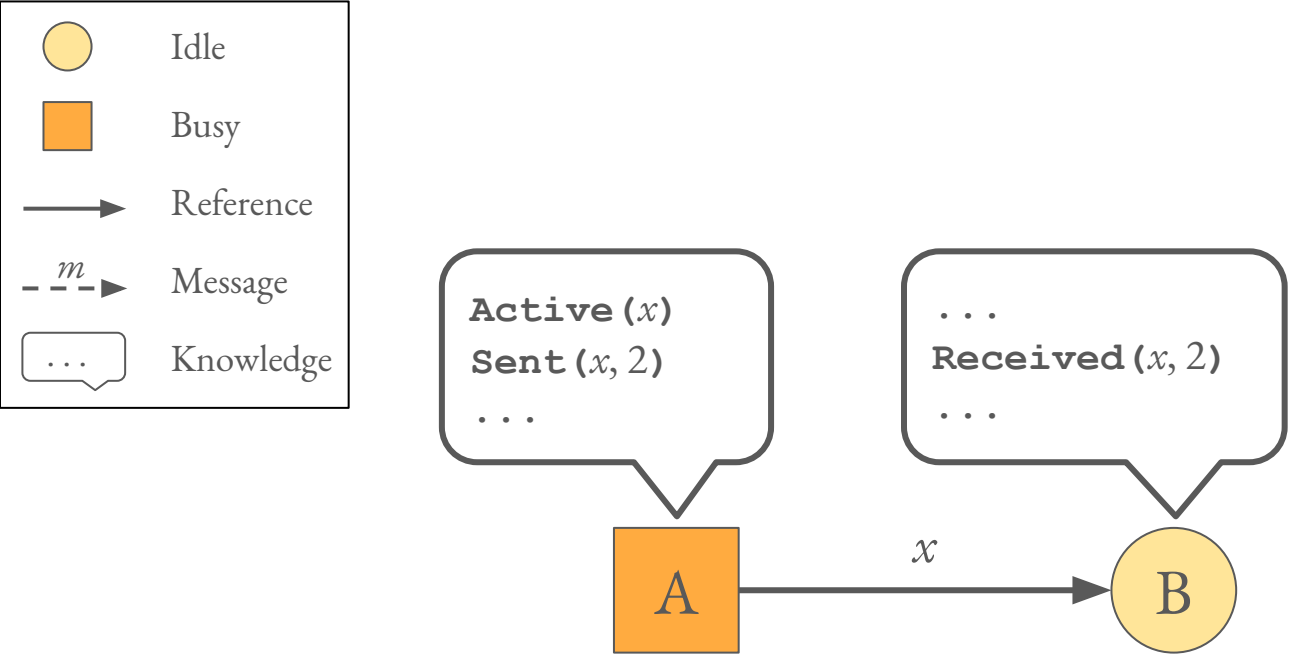

Active(x)
Sent(x, 1)
. . .
. . .
Received(x, 1)
. . .
A
x
B

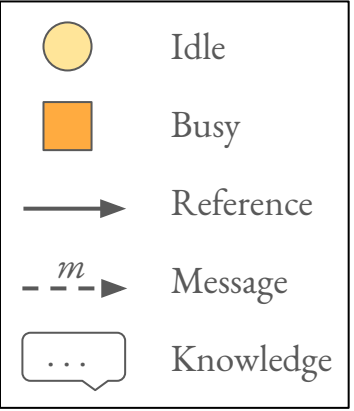

Idle
Busy
Reference
m
Message
Knowledge

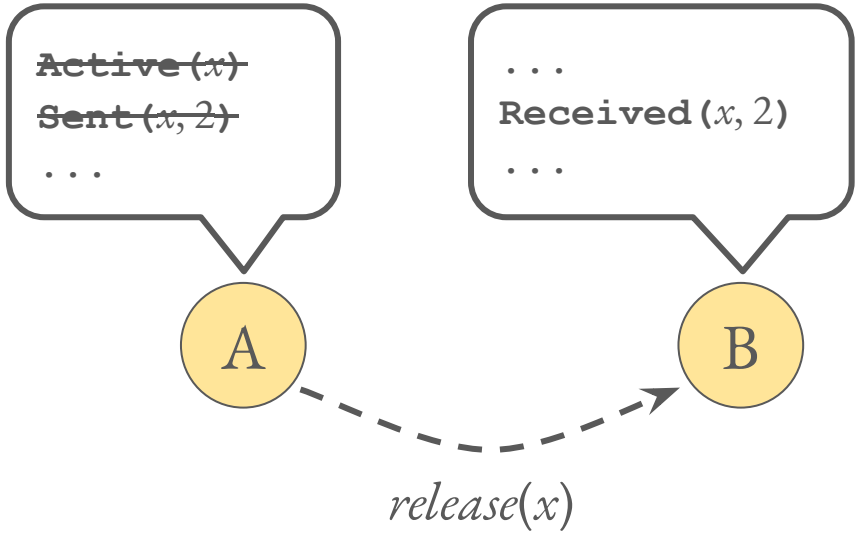

Active(x)
Sent(x, 2)
...
...
Received(x, 1)
...
A
x
B
info(x, y)

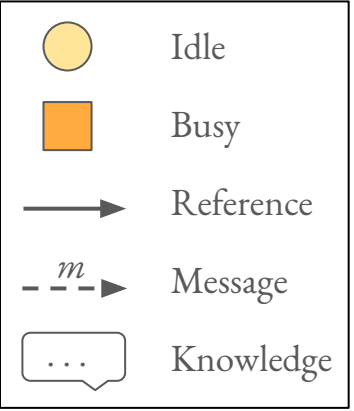

Idle
Busy
Reference
m
Message
Knowledge

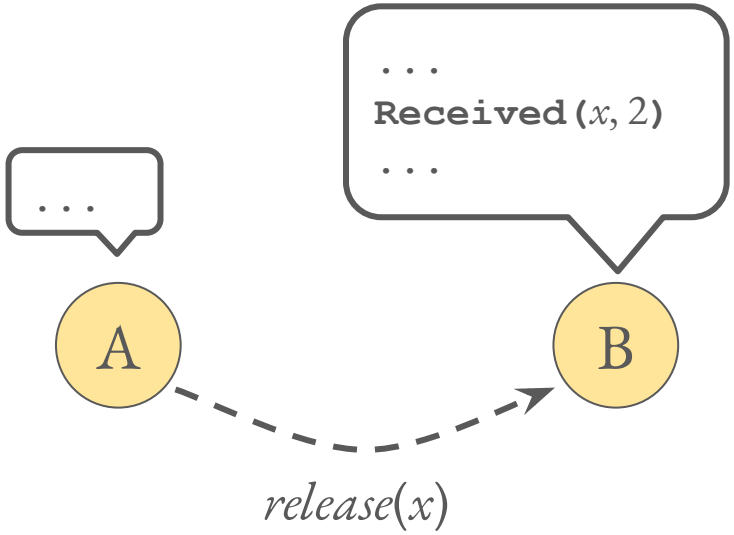

Active(x)
Sent(x, 2)
...
...
Received(x, 2)
...
A
x
B

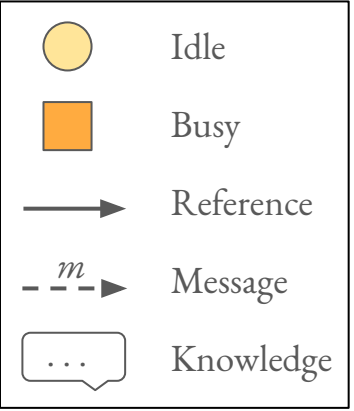

Idle
Busy
Reference
m
Message
. . .
Knowledge
Active(x)
Sent(x, 2)
. . .
. . .
Received(x, 2)
. . .
A
x
B

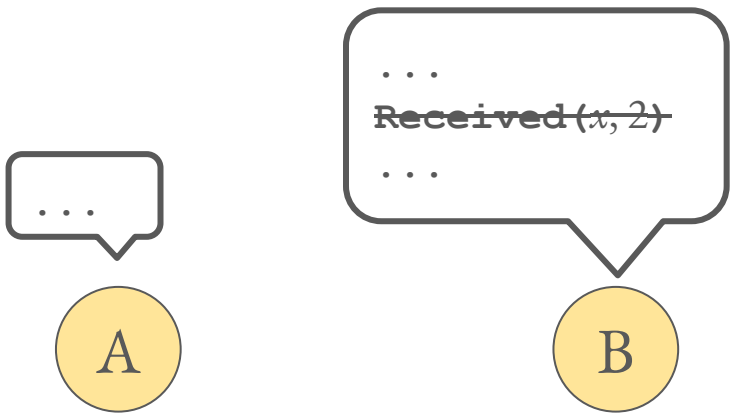

Idle
Busy
Reference
m
Message
. . .
Knowledge

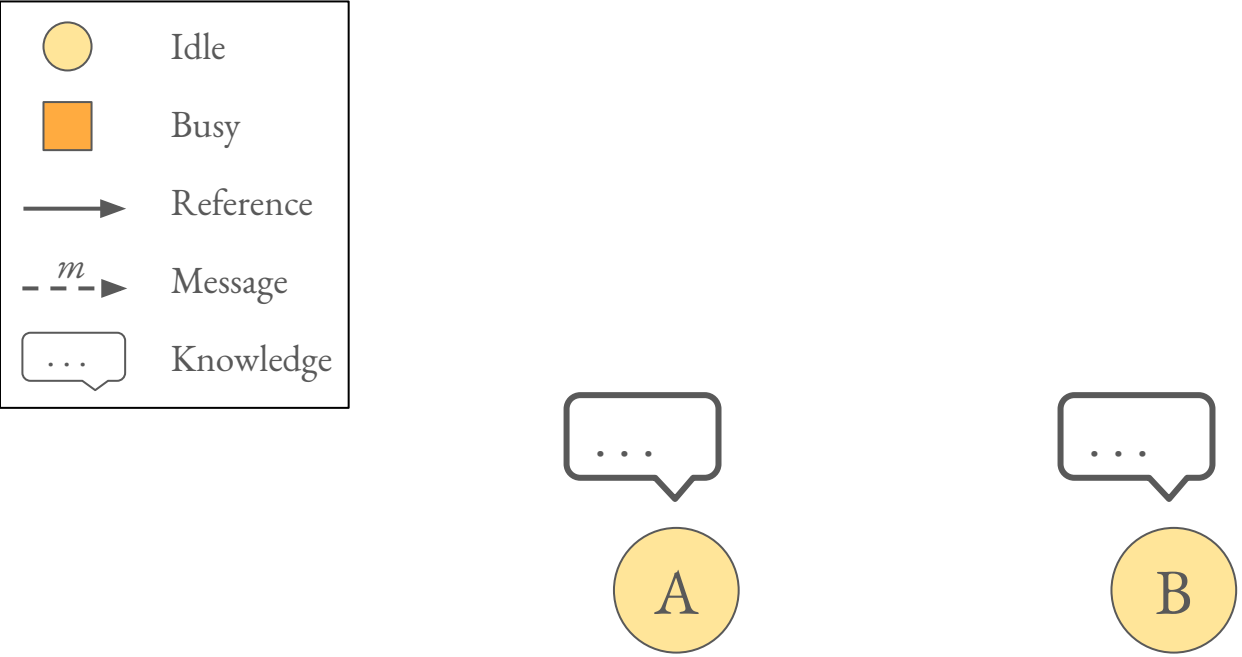

Active(x)
Sent(x, 2)
. . .
. . .
Received(x, 2)
. . .
A
B
release(x)


Idle
Busy
Reference
m
Message
. . .
Knowledge


. . .
Received(x, 2)
. . .
. . .
A
B
release(x)


Idle
Busy
Reference
m
Message
Knowledge


. . .
A
. . .
Received (x, 2)
. . .
B


Idle
Busy
Reference
m
Message
Knowledge
A
B

Part V: Termination Detection

“Appear to be terminated”?

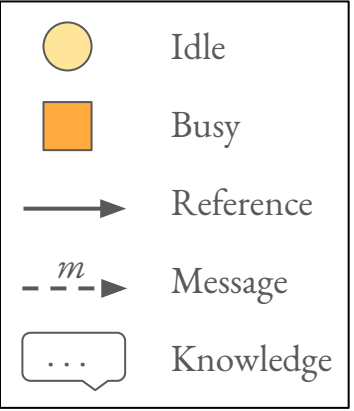
“Appear to be terminated”?

An actor is **terminated** if it is blocked…
and its potential inverse acquaintances are blocked…
and their potential inverse acquaintances are blocked…
and so on.

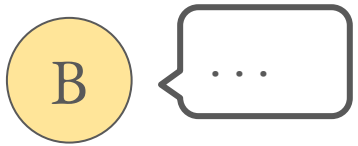
“Appear to be terminated”?

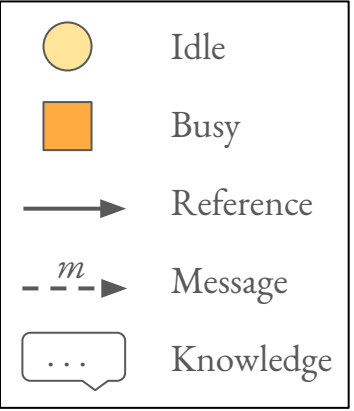
Let *S* be a set of snapshots.

Assume B is the first actor in *S* to take a snapshot.



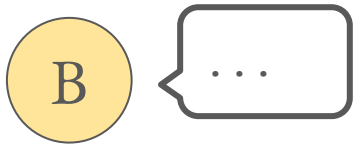
“Appear to be terminated”?

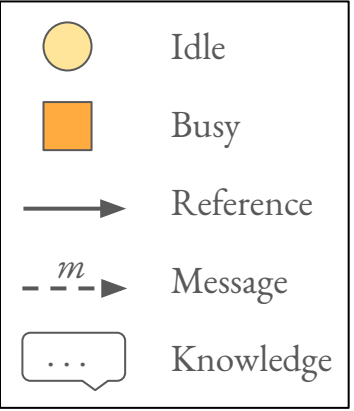




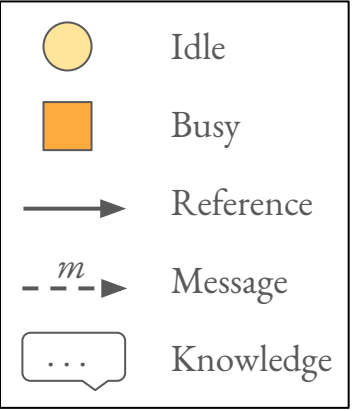
“Appear to be terminated”?

If B has no facts **`Created`**`(`x : A → B`)` then it has no potential inverse acquaintances (see paper).



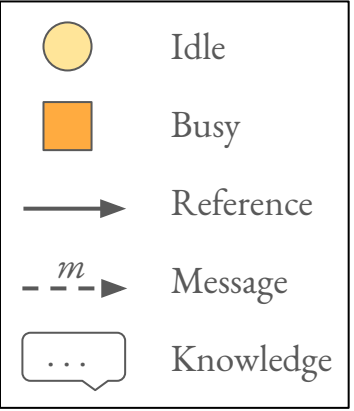


“Appear to be terminated”?



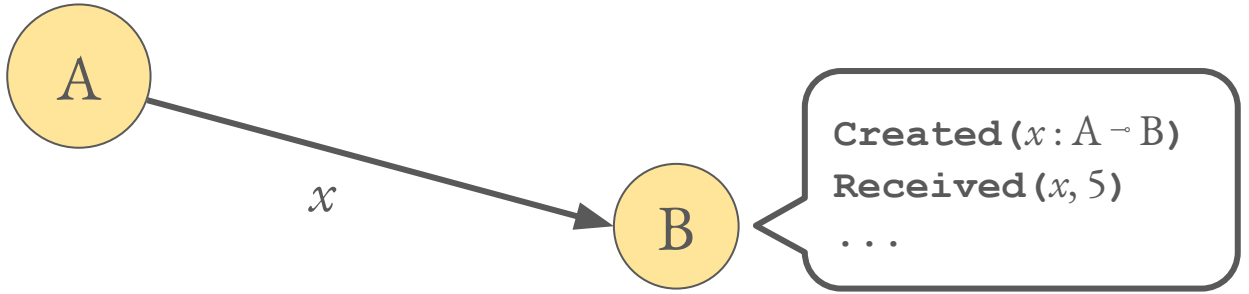
“Appear to be terminated”?

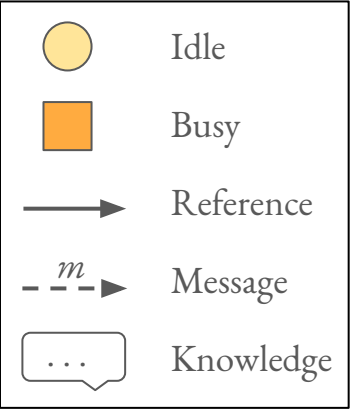
To find out if B is terminated, we need a snapshot from A.



“Appear to be terminated”?

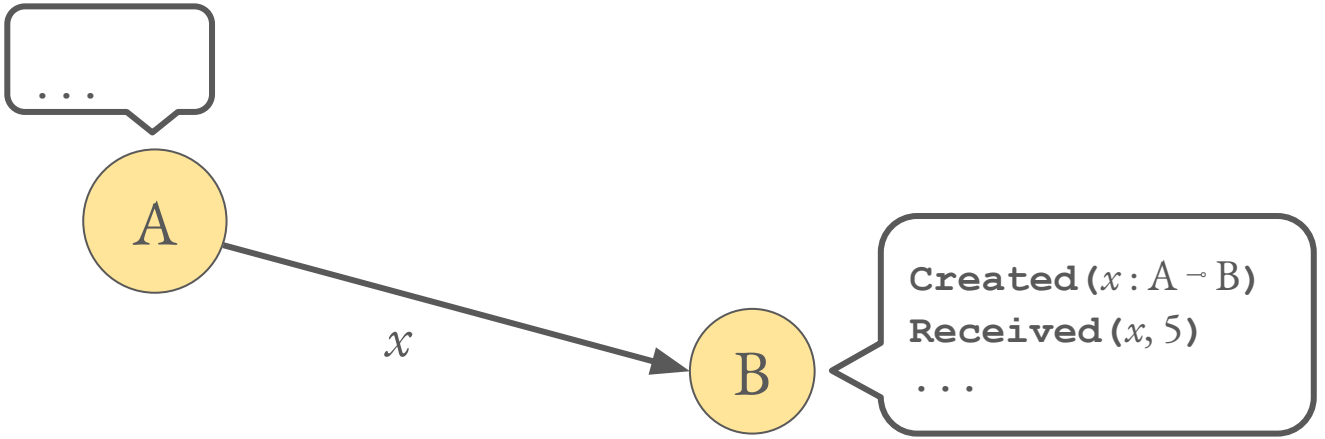
To find out if B is terminated, we need a snapshot from A.

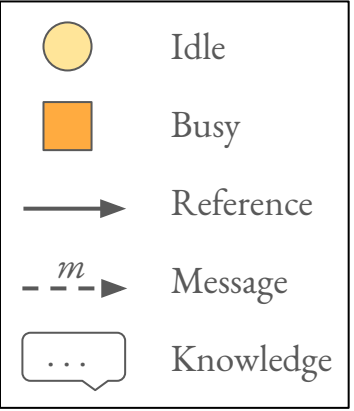




“Appear to be terminated”?

What if A’s snapshot doesn’t contain **`Active`**(*x*)?

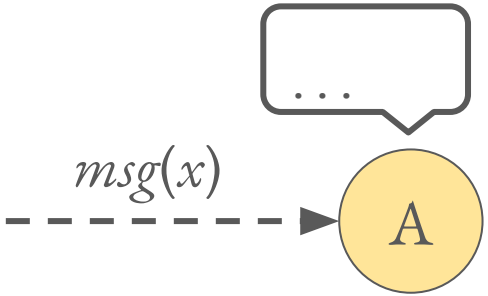


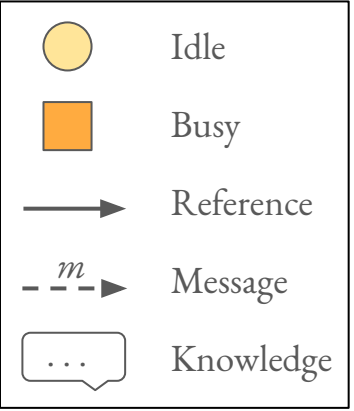


“Appear to be terminated”?

What if A’s snapshot doesn’t contain **`Active(`***x***`)`**?

Case 1: It hasn’t received *x* yet.



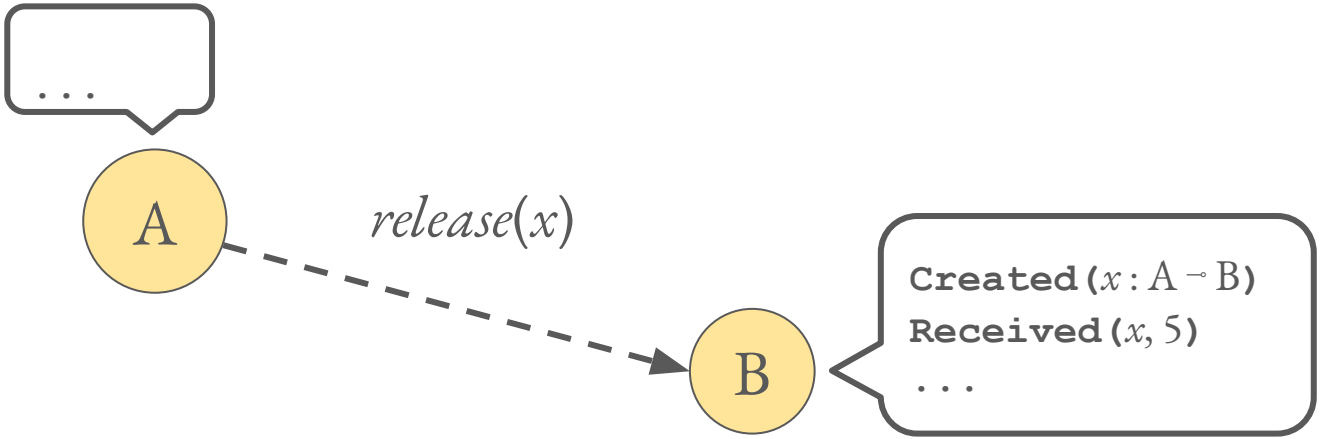


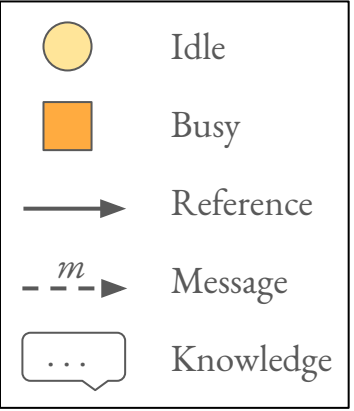
“Appear to be terminated”?

What if A’s snapshot doesn’t contain **`Active`**`(`x`)`?

Case 1: It hasn’t received x yet.

Case 2: It has released x already.





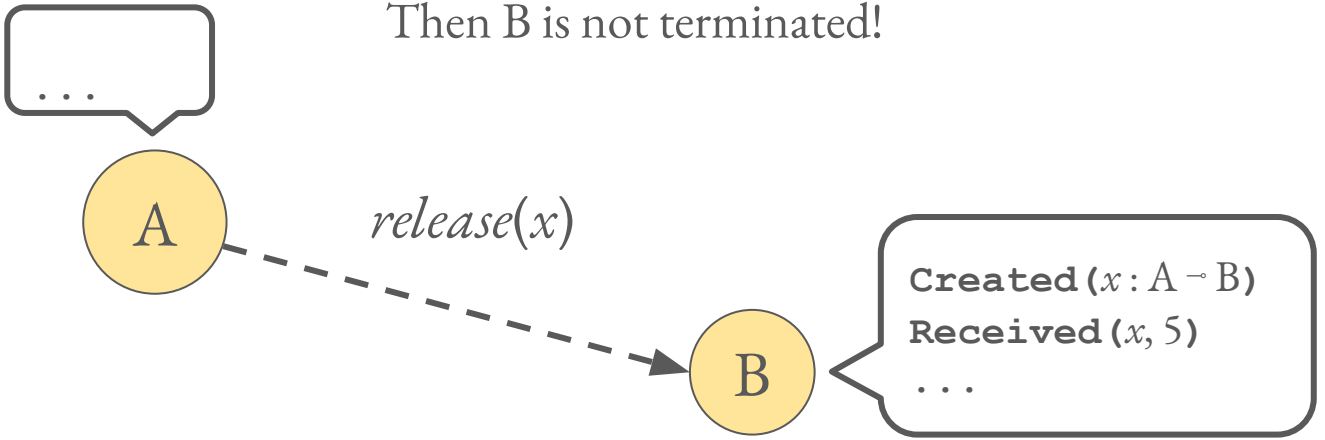
“Appear to be terminated”?

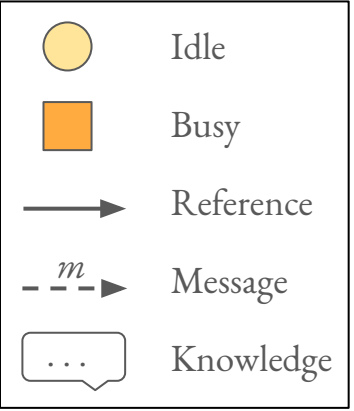
What if A’s snapshot doesn’t contain **`Active(`***x***`)`**?

Case 1: It hasn’t received *x* yet.

Case 2: It has released *x* already.

Then B is not terminated!

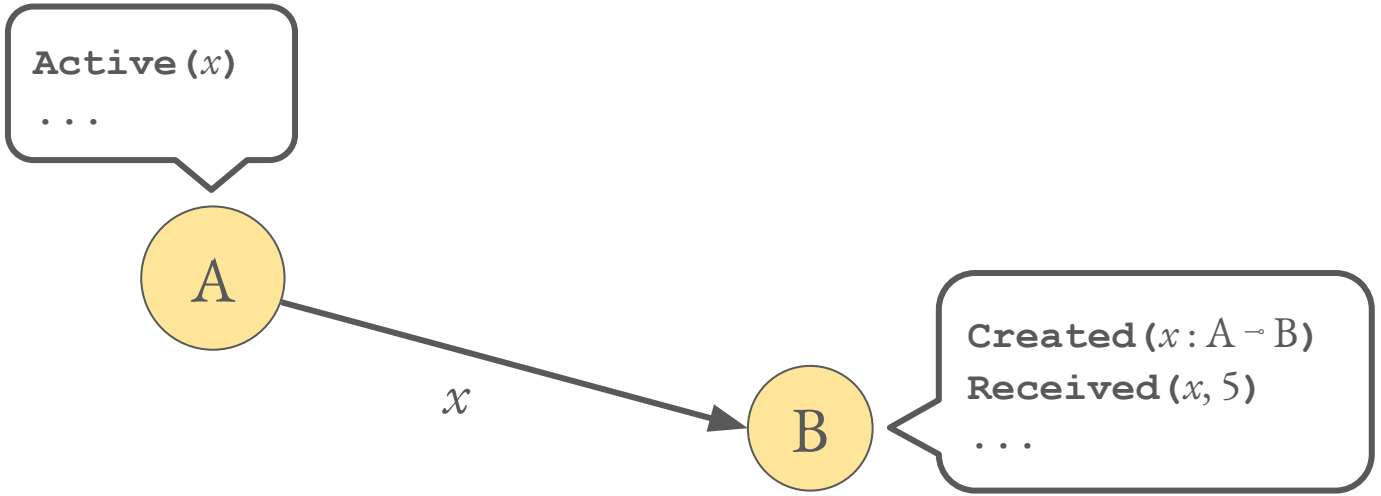


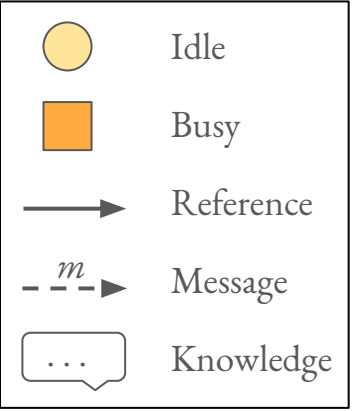


“Appear to be terminated”?

B is only terminated if...

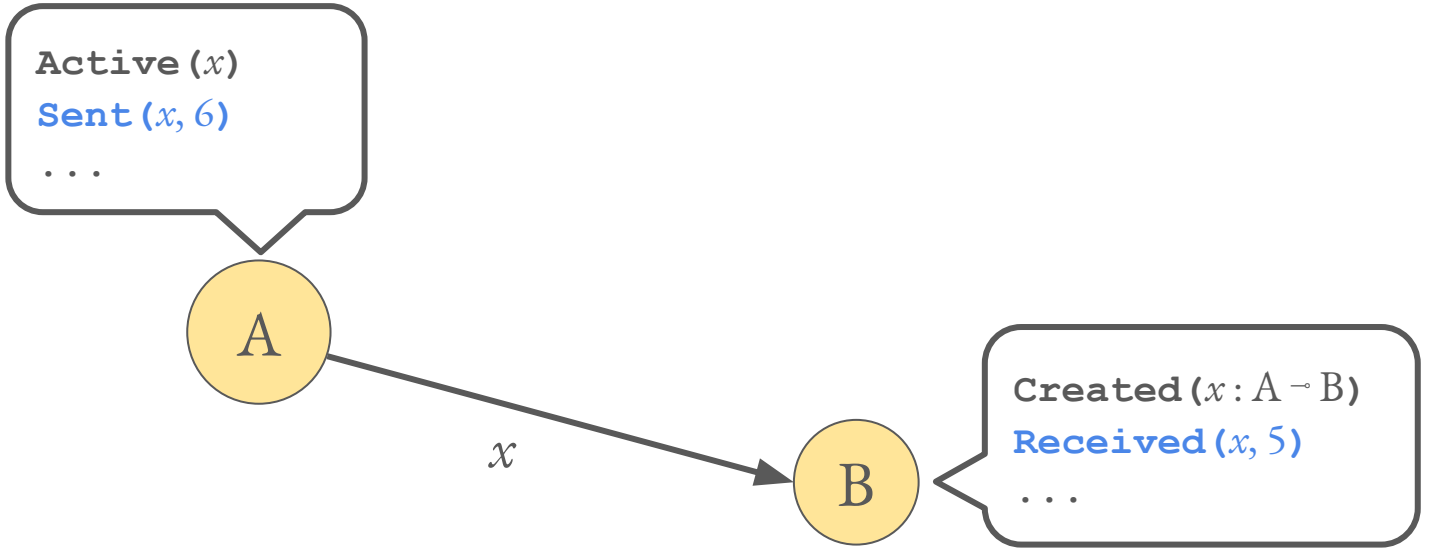
- A’s snapshot contains **`Active`**(x).

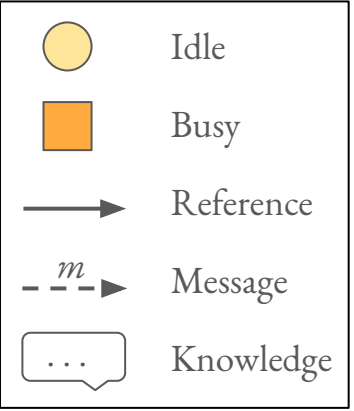




“Appear to be terminated”?

What if the message counts differ?



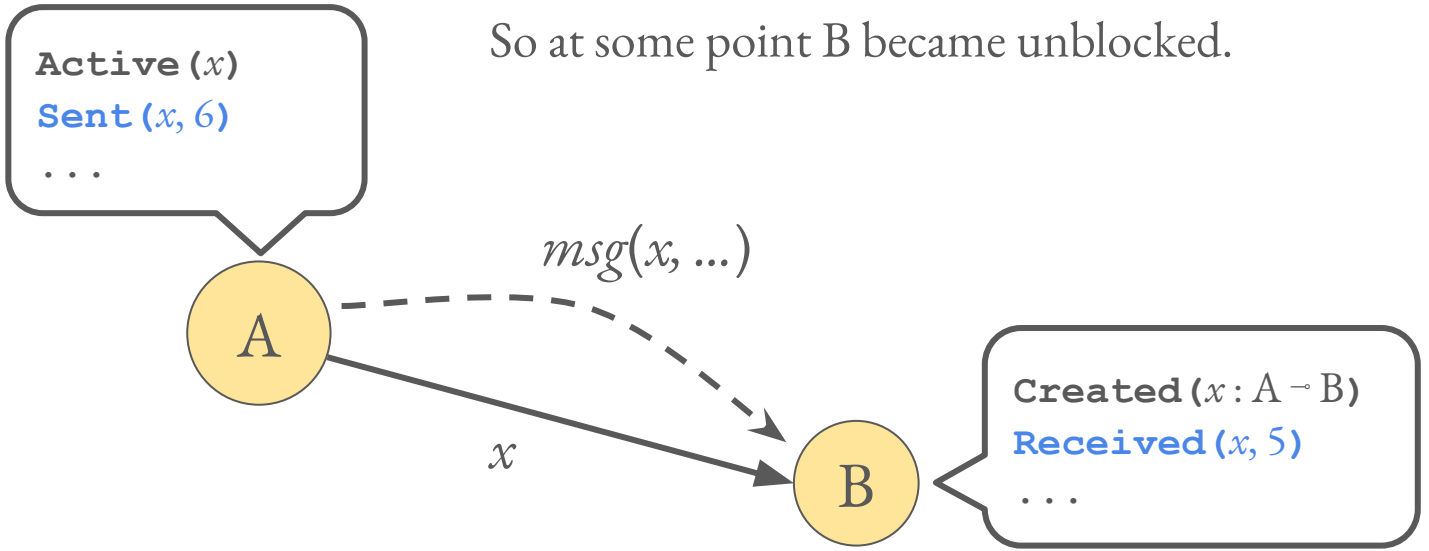


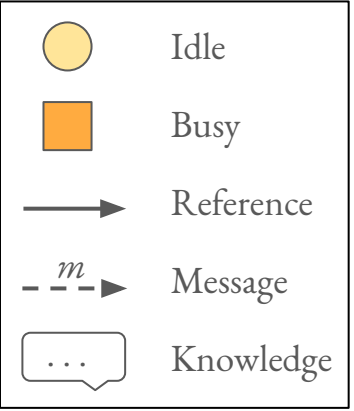
“Appear to be terminated”?

What if the message counts differ?

Recall B’s snapshot came first.

So at some point B became unblocked.

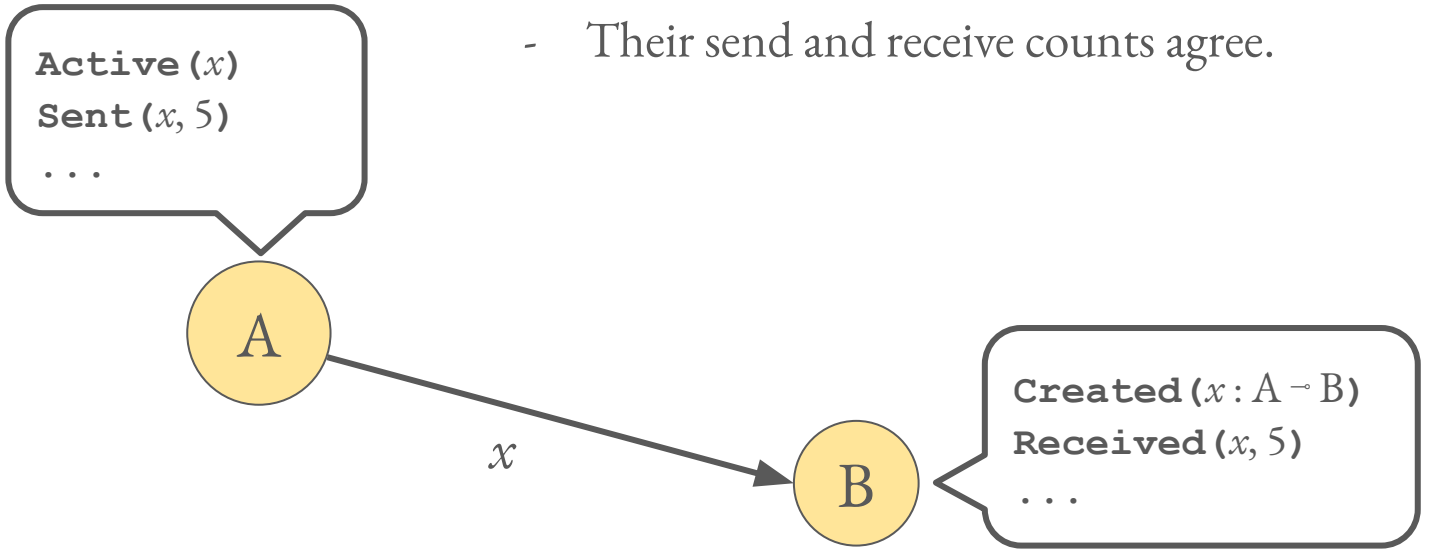


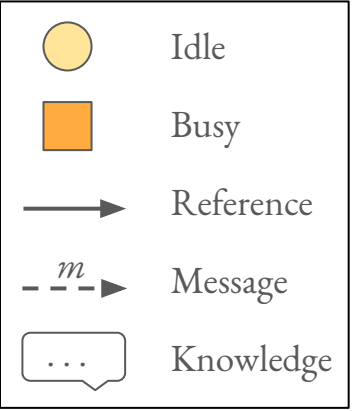


“Appear to be terminated”?

B is only terminated if...

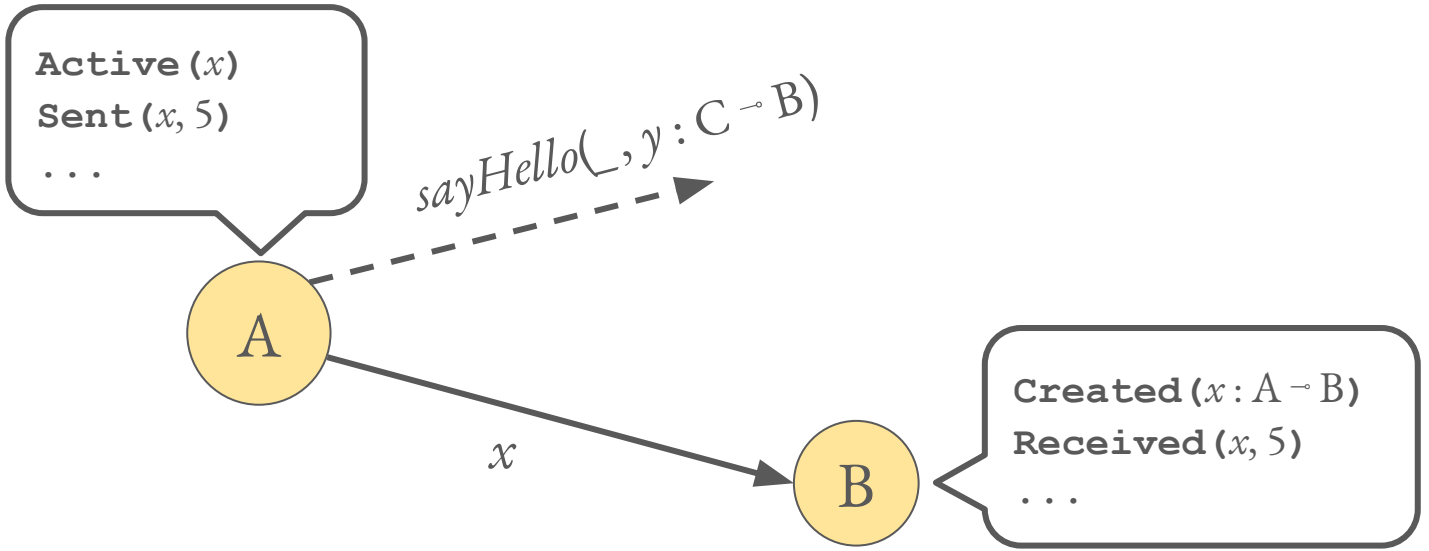
- A’s snapshot contains **`Active`**(x);
- Their send and receive counts agree.

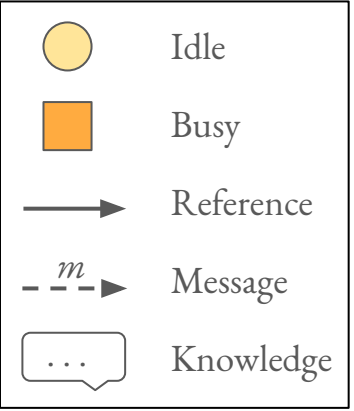




“Appear to be terminated”?

What if A created a new refob $y : C \multimap B$ before snapshot?

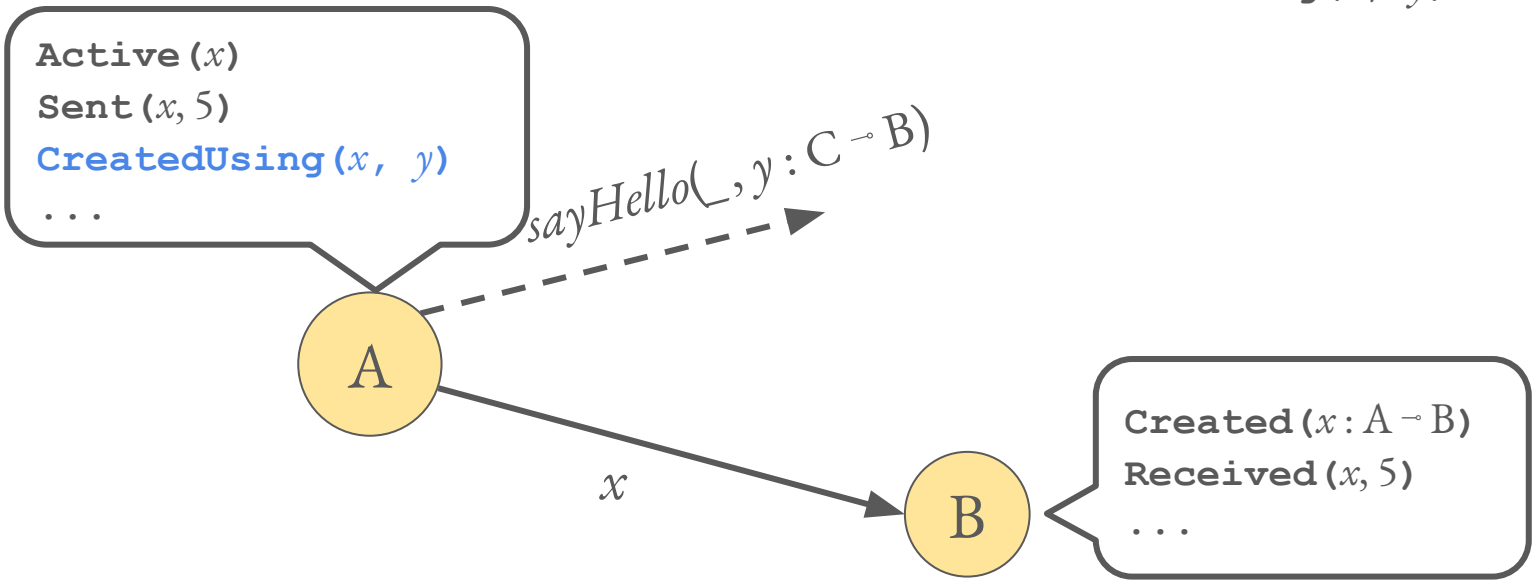


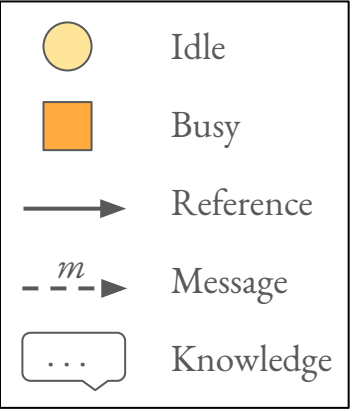


“Appear to be terminated”?

What if A created a new refob $y : C \multimap B$ before snapshot?

Then A created a fact **`CreatedUsing`**(x, y).



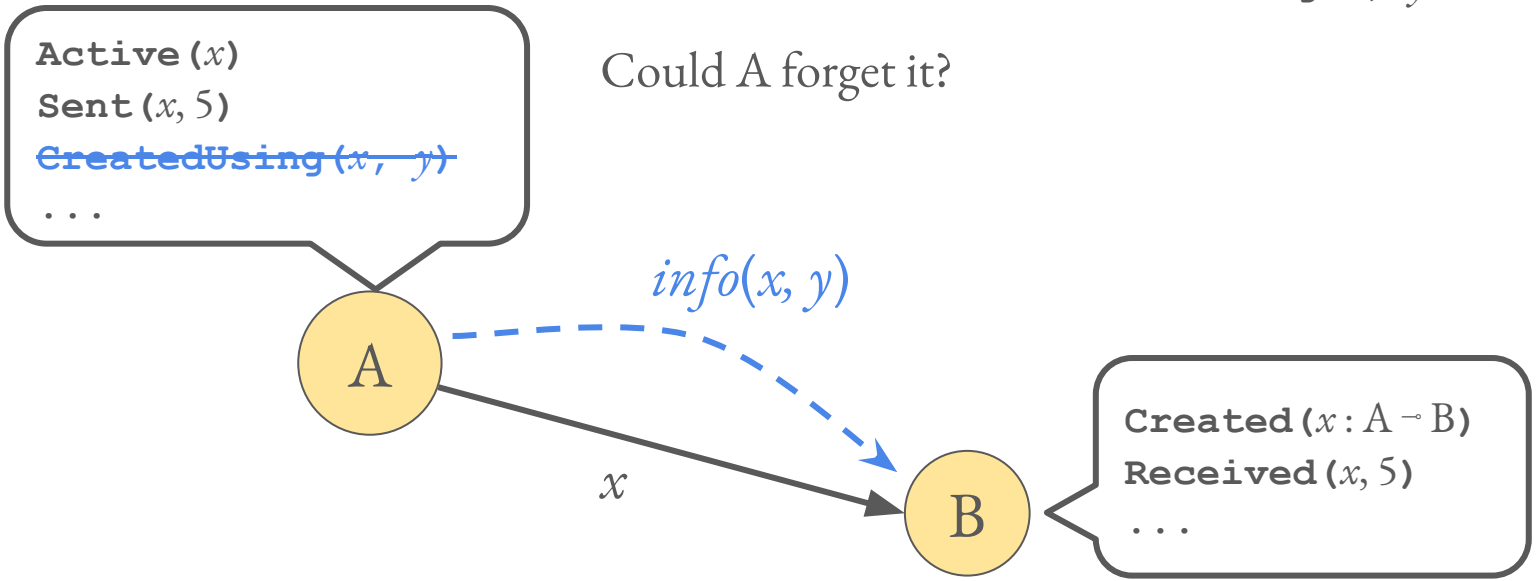


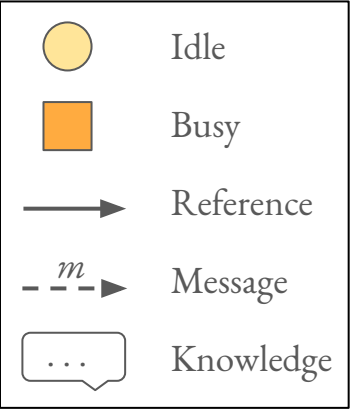
“Appear to be terminated”?

What if A created a new refob $y : C \multimap B$ before snapshot?

Then A created a fact `CreatedUsing(`x`, `y`)`.

Could A forget it?



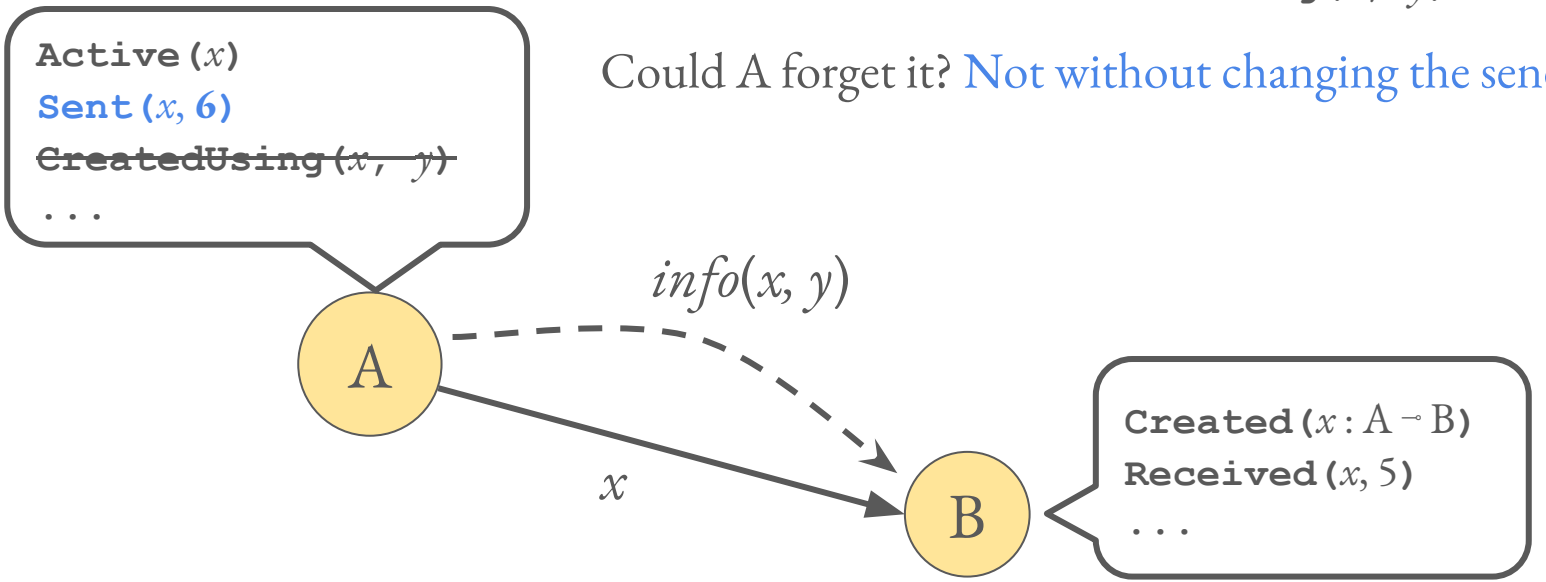


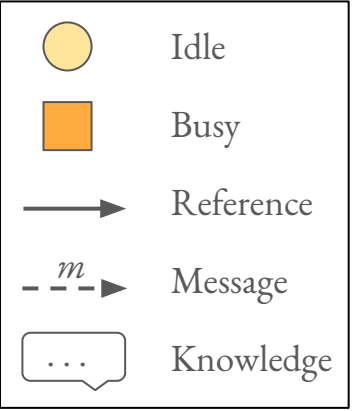
"Appear to be terminated"?

What if A created a new refob $y : C \multimap B$ before snapshot?

Then A created a fact `CreatedUsing(`x`, `y`)`.

Could A forget it? Not without changing the send count!



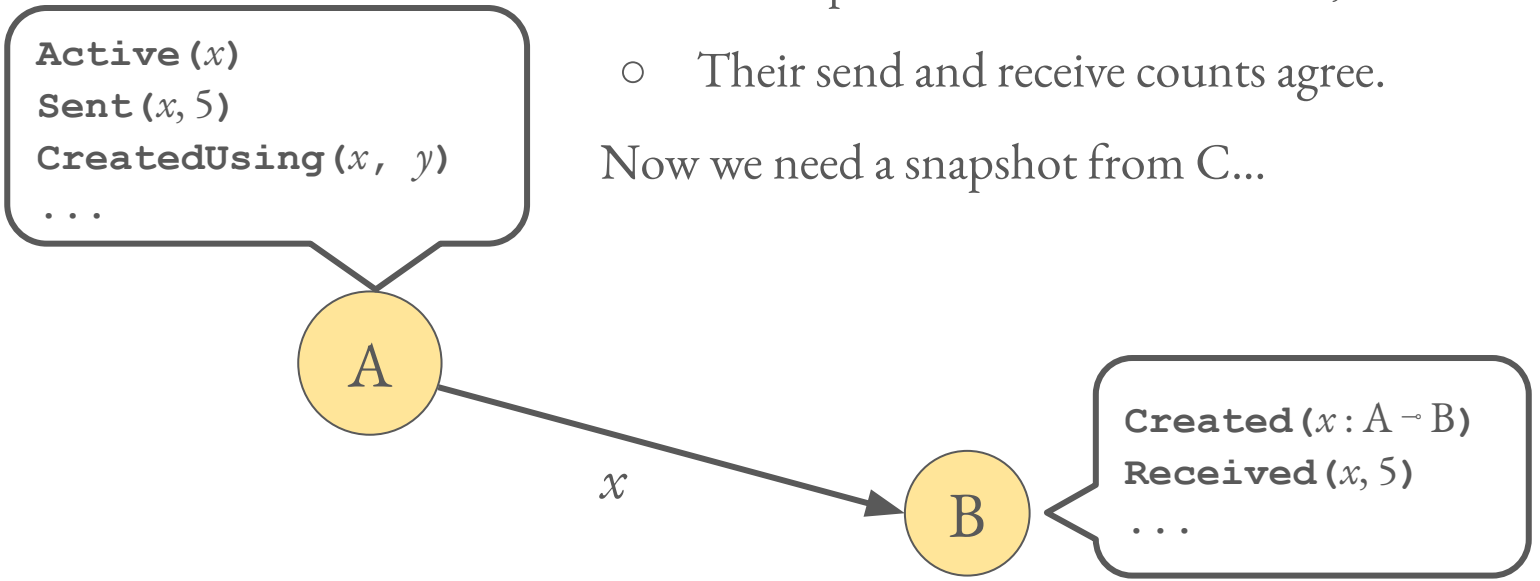


“Appear to be terminated”?

B is only terminated if...

- A’s snapshot contains **`Active`**`(`x`)`;
- Their send and receive counts agree.

Now we need a snapshot from C...



“Appear to be terminated”?

Assuming B is blocked, show that the next actor is also blocked.

“Appear to be terminated”?

Assuming B is blocked, show that the next actor
is also blocked.

Hence every actor that can potentially reach B is blocked!

Acknowledgments

This work was supported in part by the National Science Foundation under Grant No. SHF 1617401, and in part by the Laboratory Directed Research and Development program at Sandia National Laboratories, a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Thanks also to Dipayan Mukherjee, Atul Sandur, Charles Kuch, Jerry Wu, Emily Hutchinson, and the anonymous referees for their valuable feedback.