Improving Asynchrony in an Active Object Model

Brian Amedro
INRIA Sophia Antipolis, France

4th INRIA - UIUC Joint Lab Workshop
November 22-24, 2010
Urbana-Champaign, IL

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

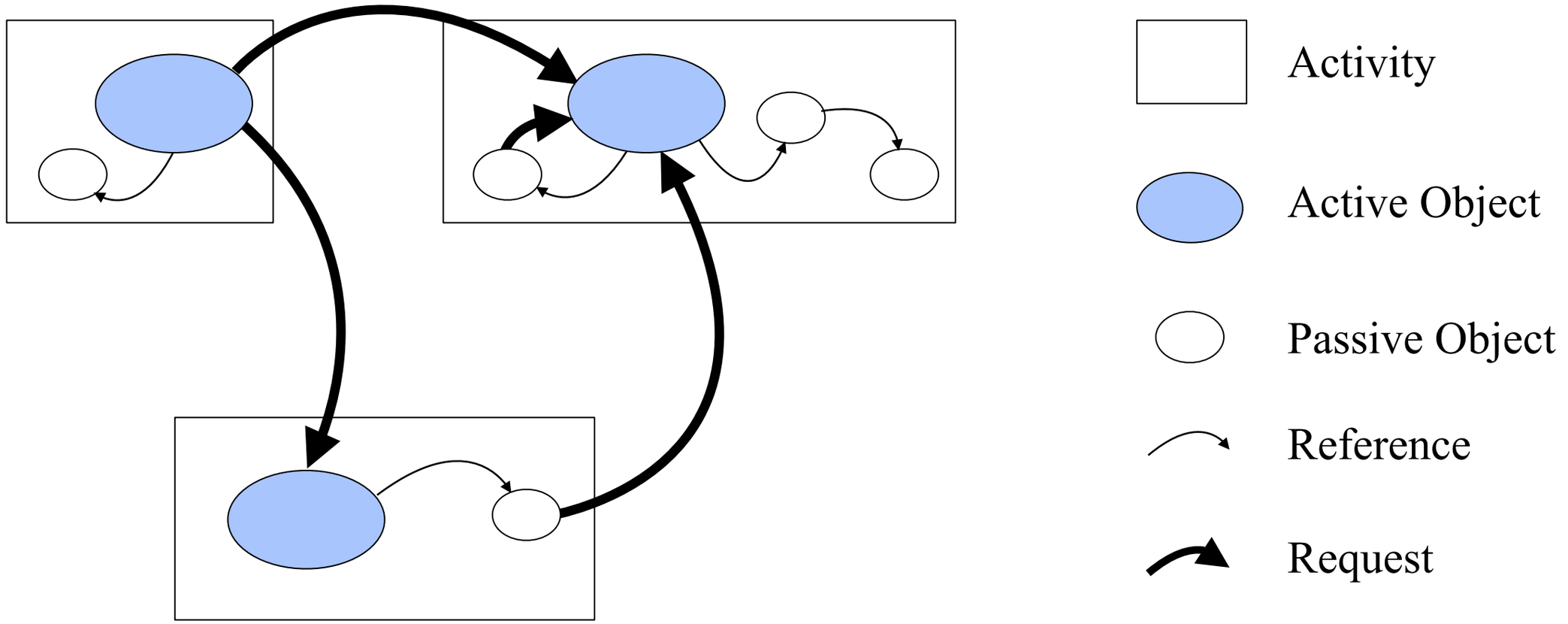
INRIA

centre de recherche
SOPHIA ANTIPOLIS - MÉDITERRANÉE

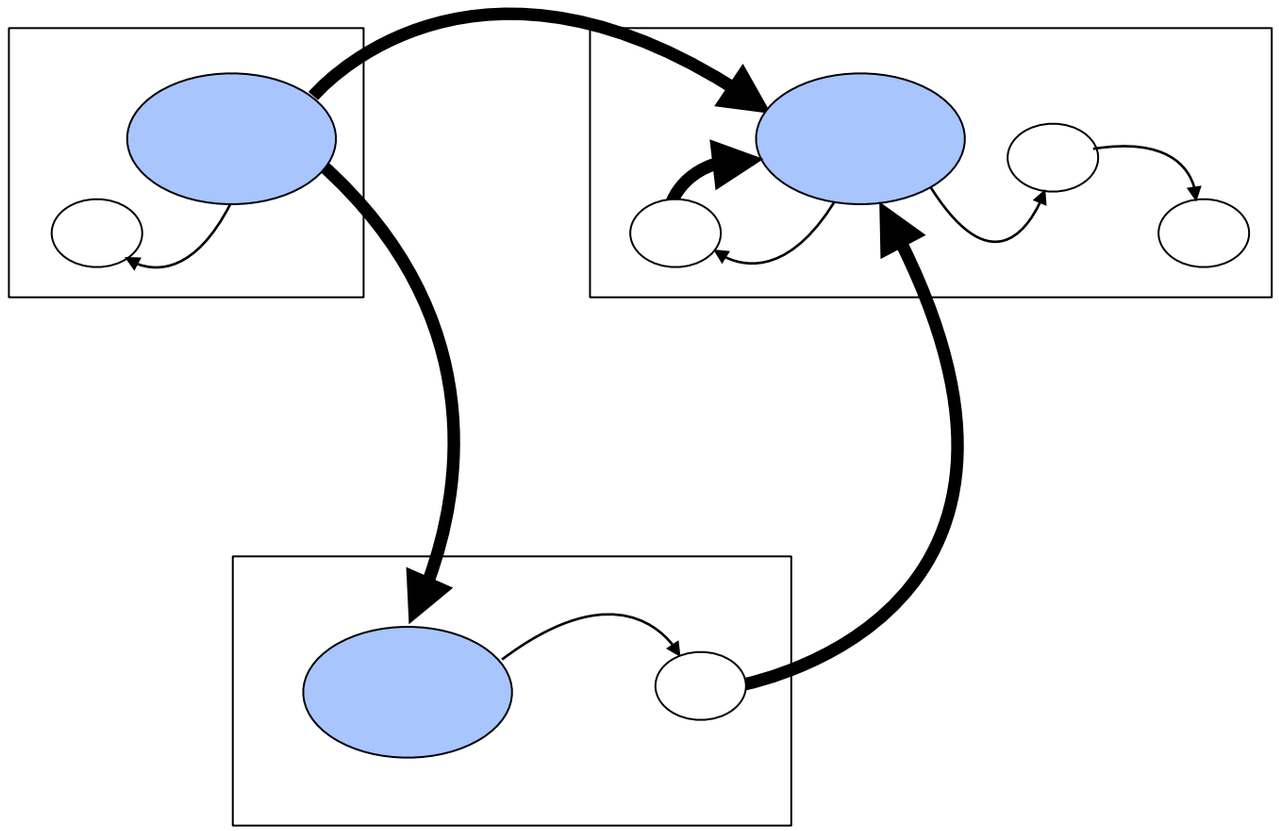
Outline

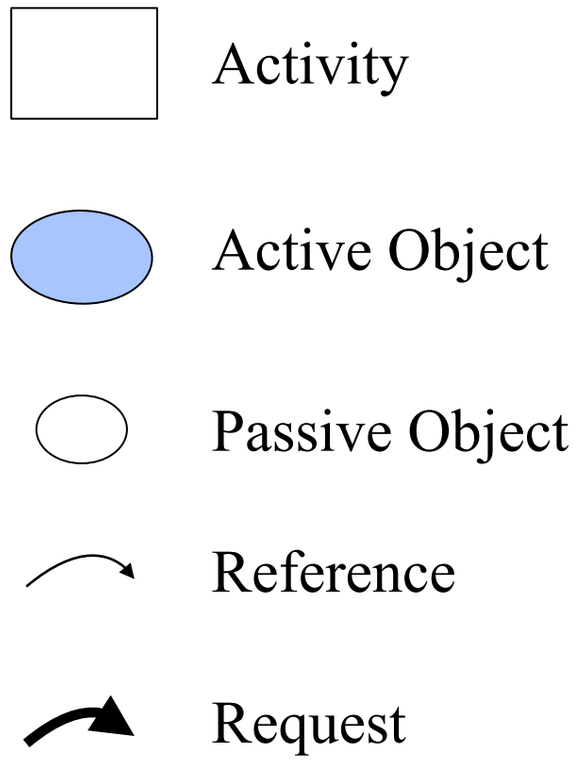
- ProActive model overview
  - active objects
  - request queue
  - rendezvous
- Characterize & manage the requests
  - forget-on-send
  - wait-by-necessity
  - sterility
- Losing rendezvous
  - algorithm

ProActive model overview
Asynchronous Sequential Processes



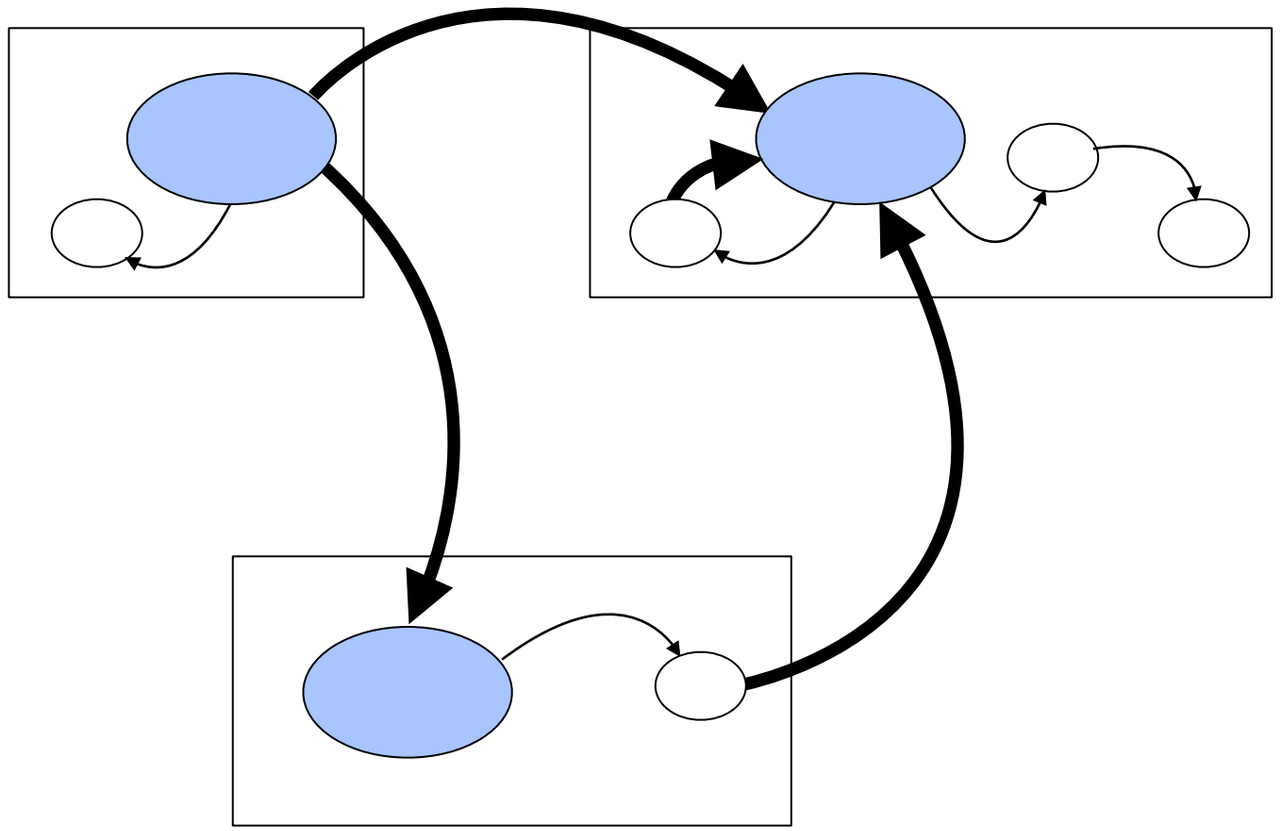
ProActive model overview
Asynchronous Sequential Processes

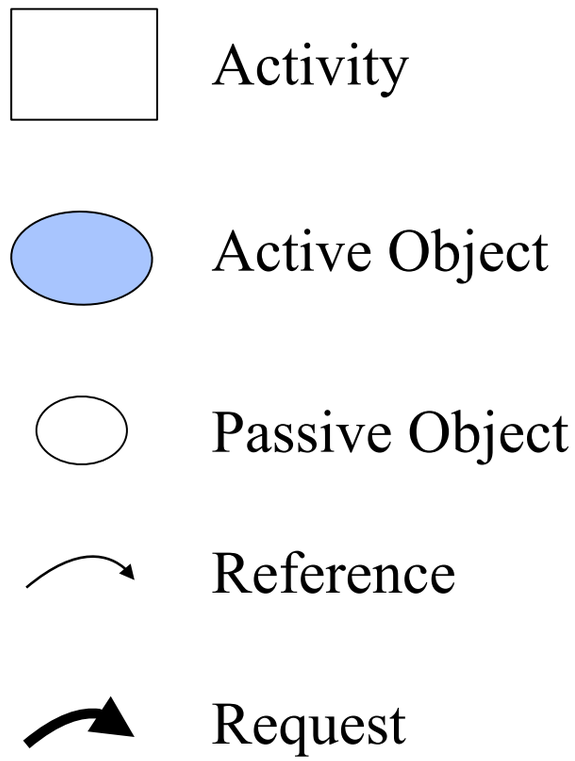




- No references across the activities
- Only requests
- No memory sharing

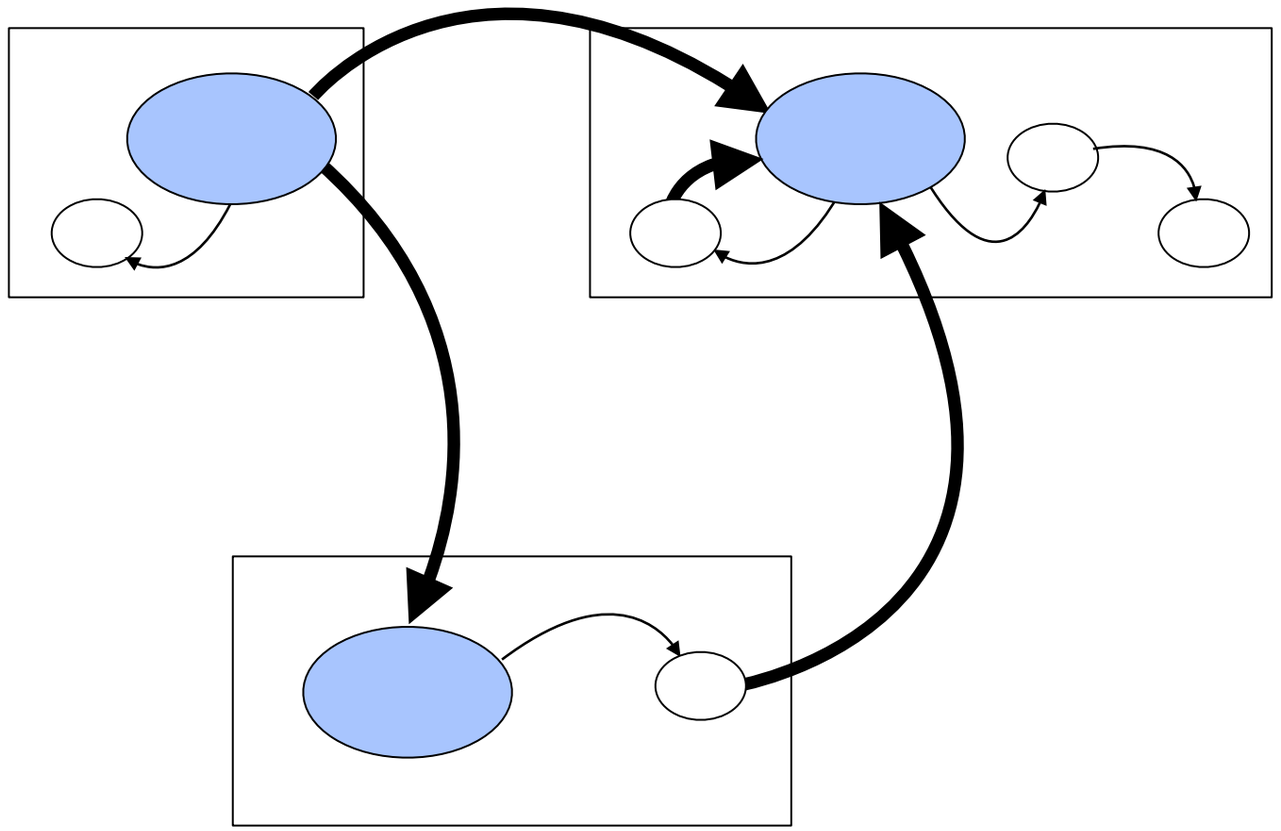
ProActive model overview
Asynchronous Sequential Processes

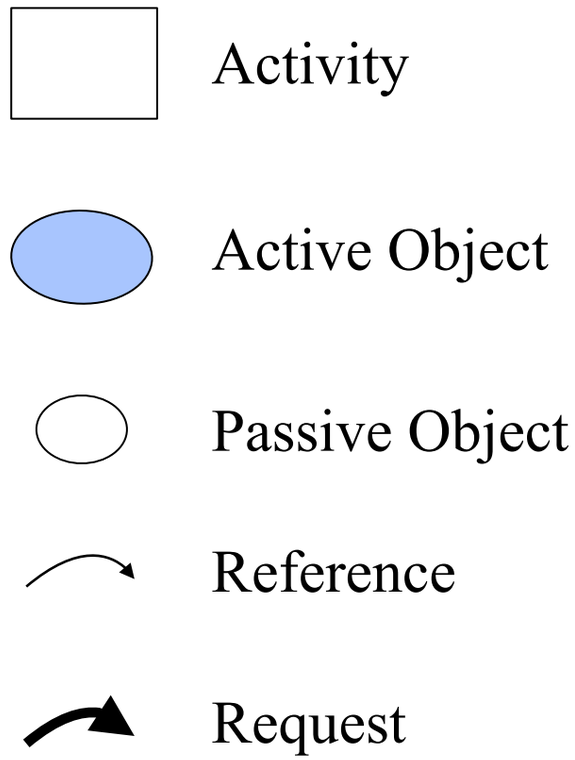




- No references across the activities
- Only requests
- No memory sharing

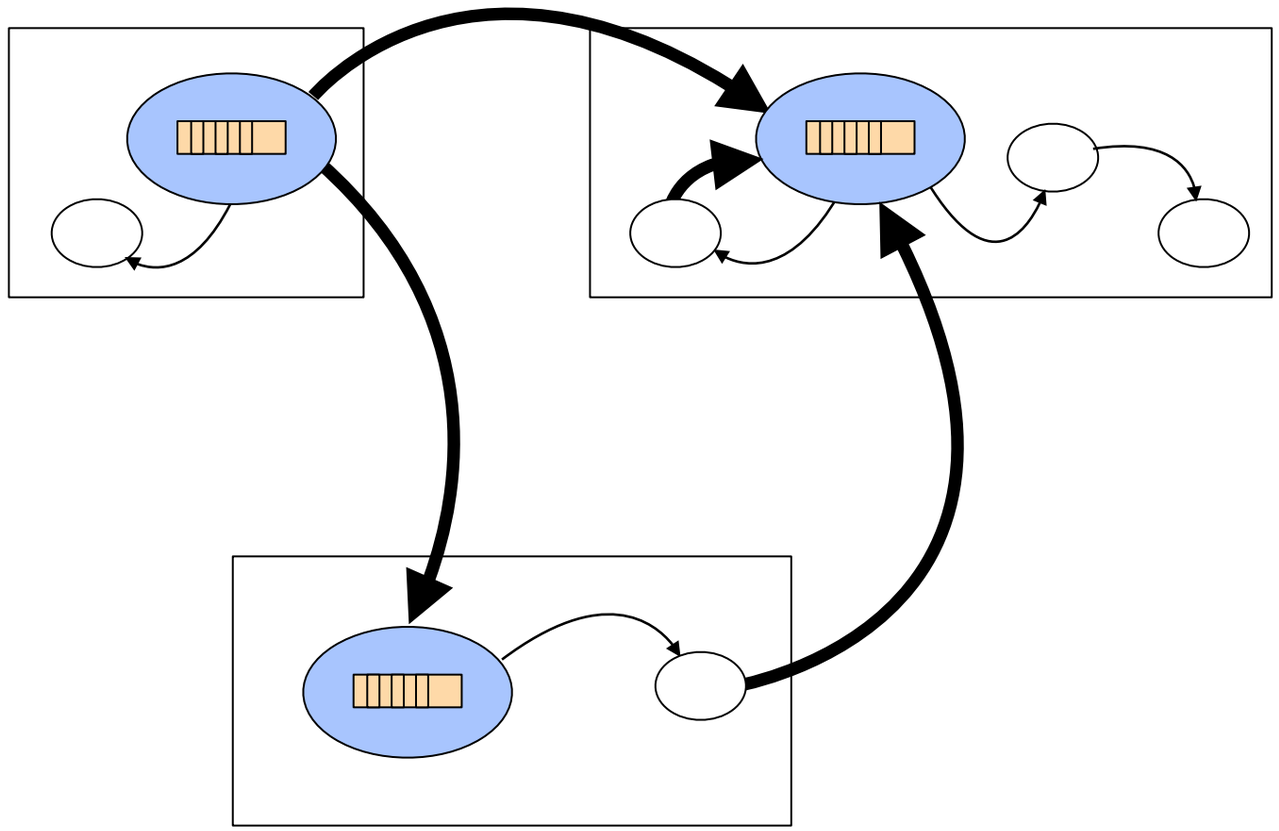
ProActive model overview
Asynchronous Sequential Processes

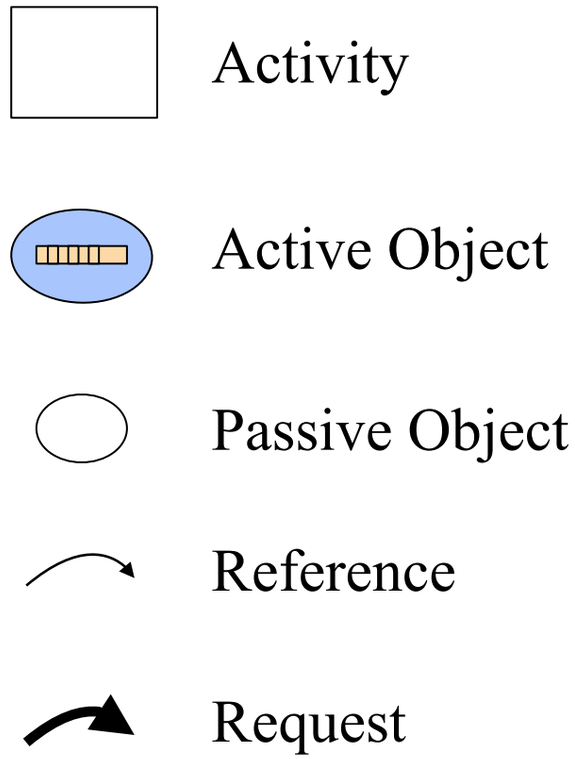




- No references across the activities
- Only requests
- No memory sharing

ProActive model overview
Asynchronous Sequential Processes

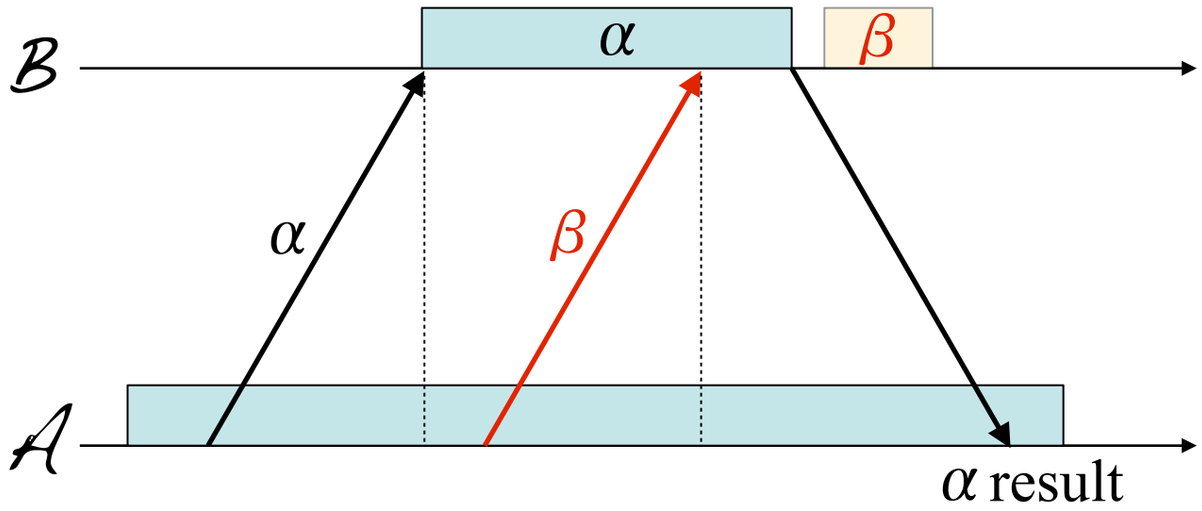




- No references across the activities
- Only requests
- No memory sharing
- Each activity has a request queue

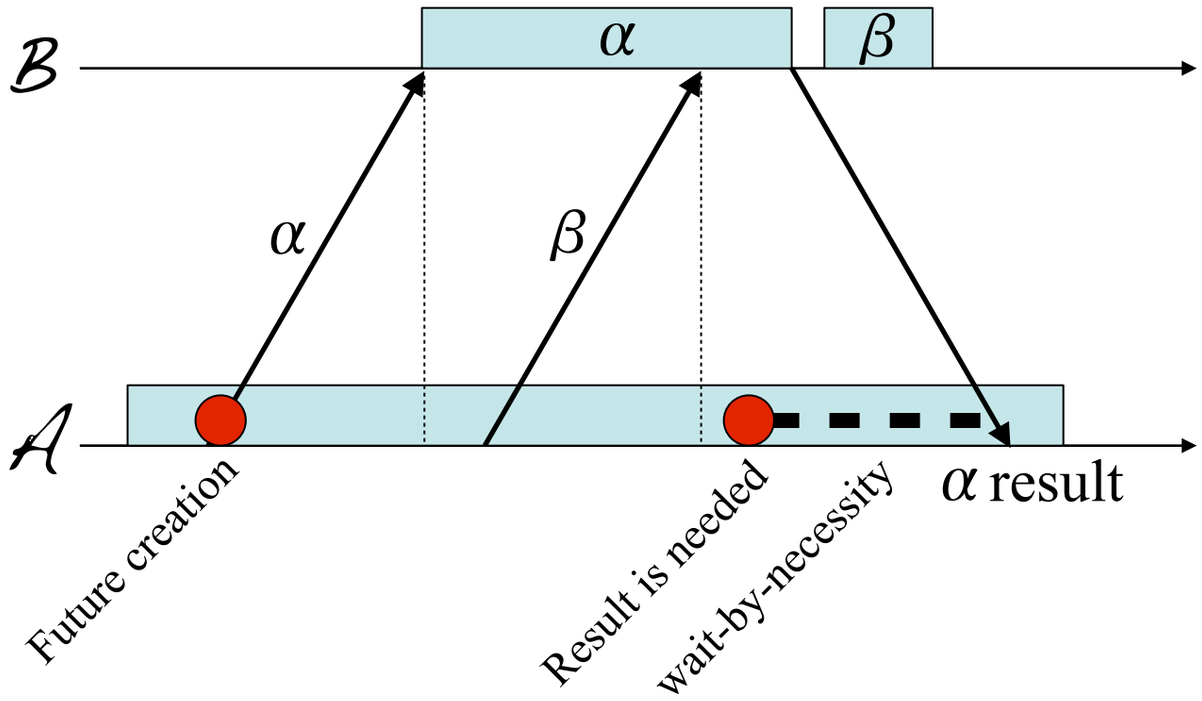
ProActive model overview
Asynchronous Sequential Processes

1. Asynchronous service
2. Future based result with wait-by-necessity
3. Blocking communication



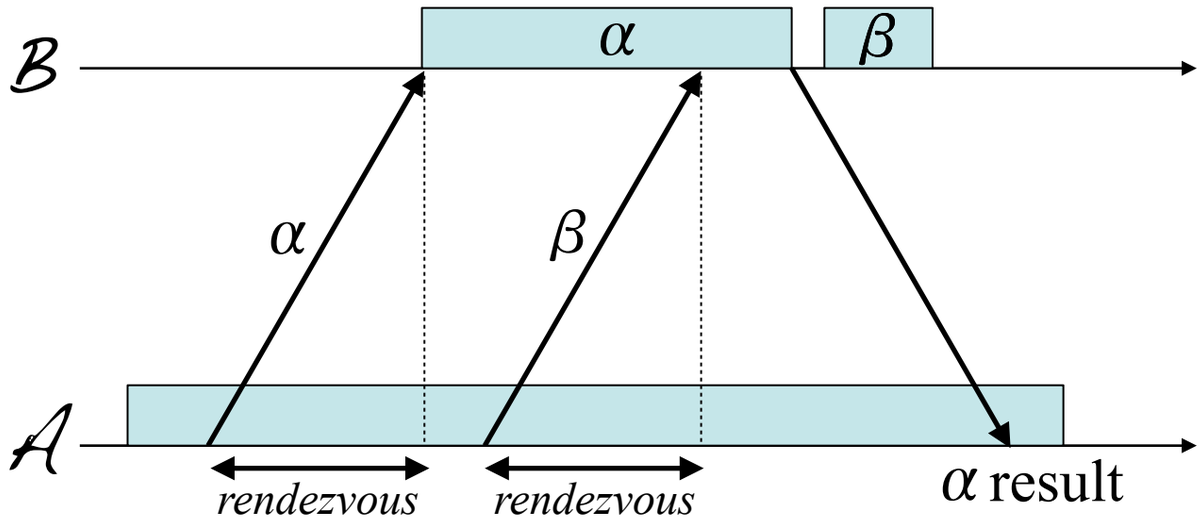
ProActive model overview
Asynchronous Sequential Processes

1. Asynchronous service
2. Future based result with wait-by-necessity
3. Blocking communication



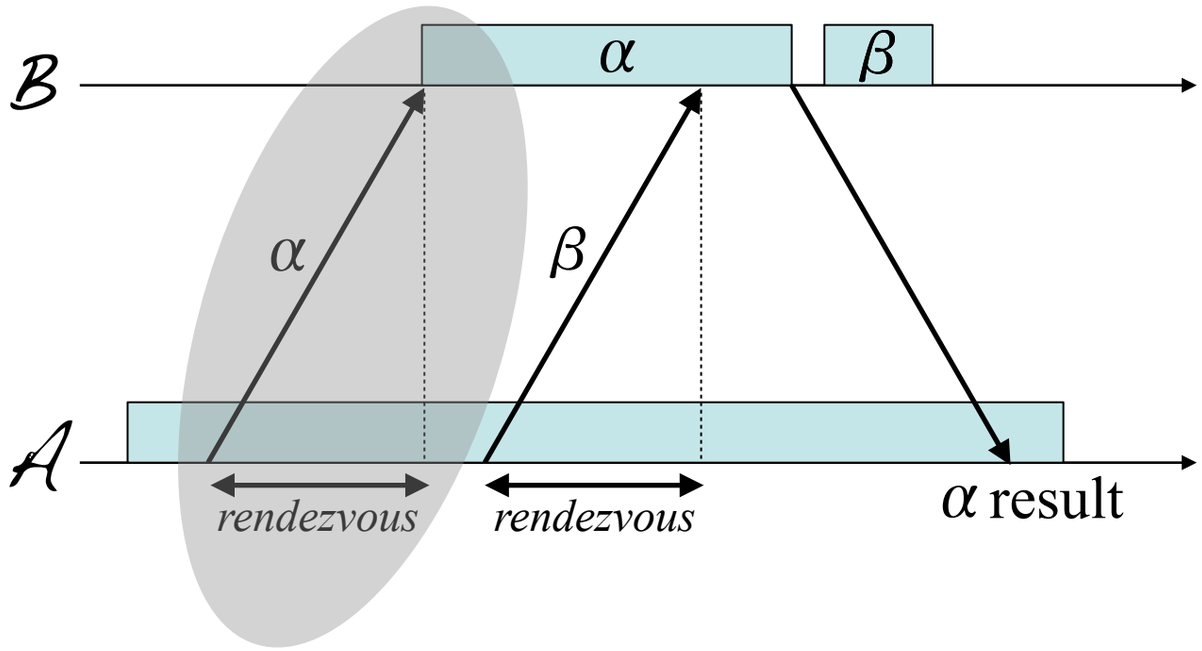
ProActive model overview
Asynchronous Sequential Processes

1. Asynchronous service
2. Future based result with wait-by-necessity
3. Blocking communication



ProActive model overview
Asynchronous Sequential Processes

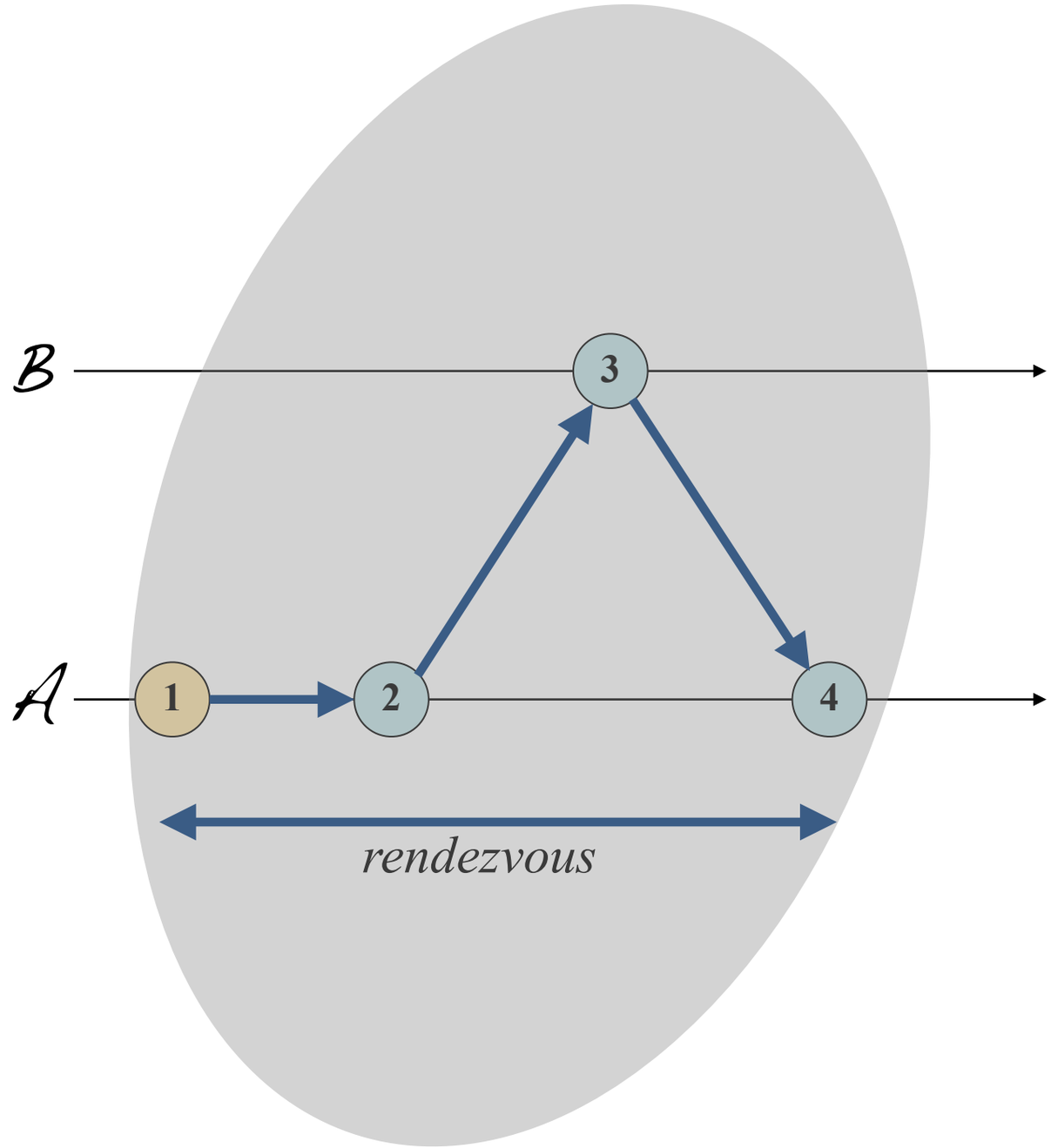
1. Asynchronous service
2. Future based result with wait-by-necessity
3. Blocking communication



ProActive model overview
Asynchronous Sequential Processes

The rendezvous relies on four steps

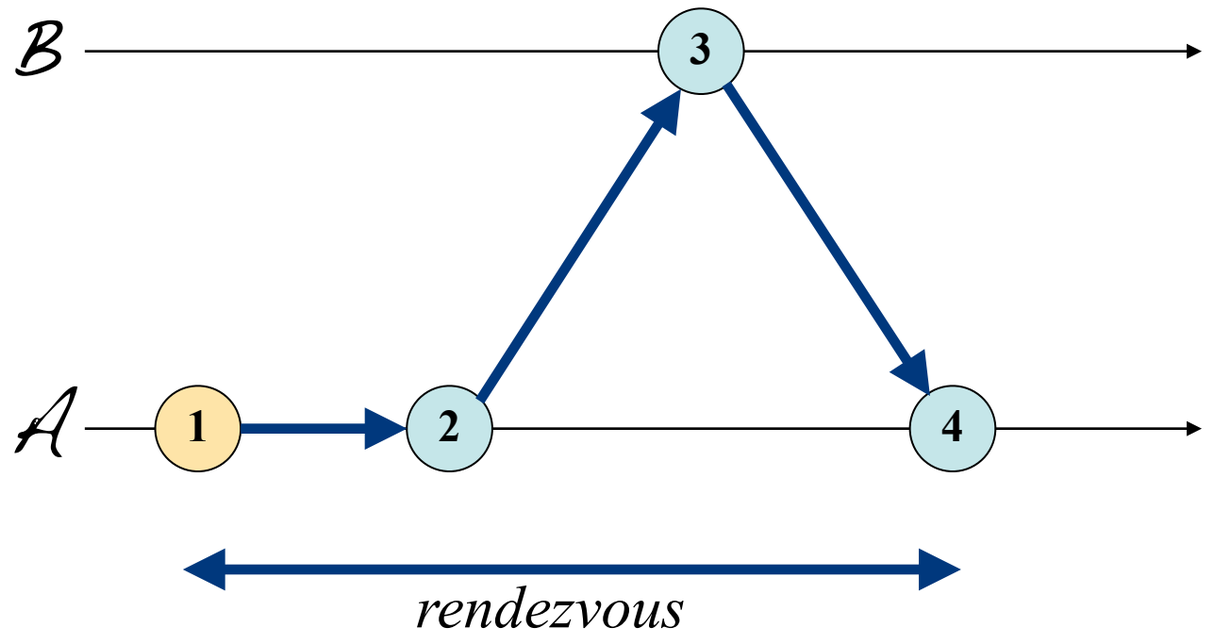
1. Request creation
2. Actual sending of the request
3. Queuing into the receiver's request queue
4. Receiving the acknowledgement



ProActive model overview
Asynchronous Sequential Processes

The rendezvous relies on four steps

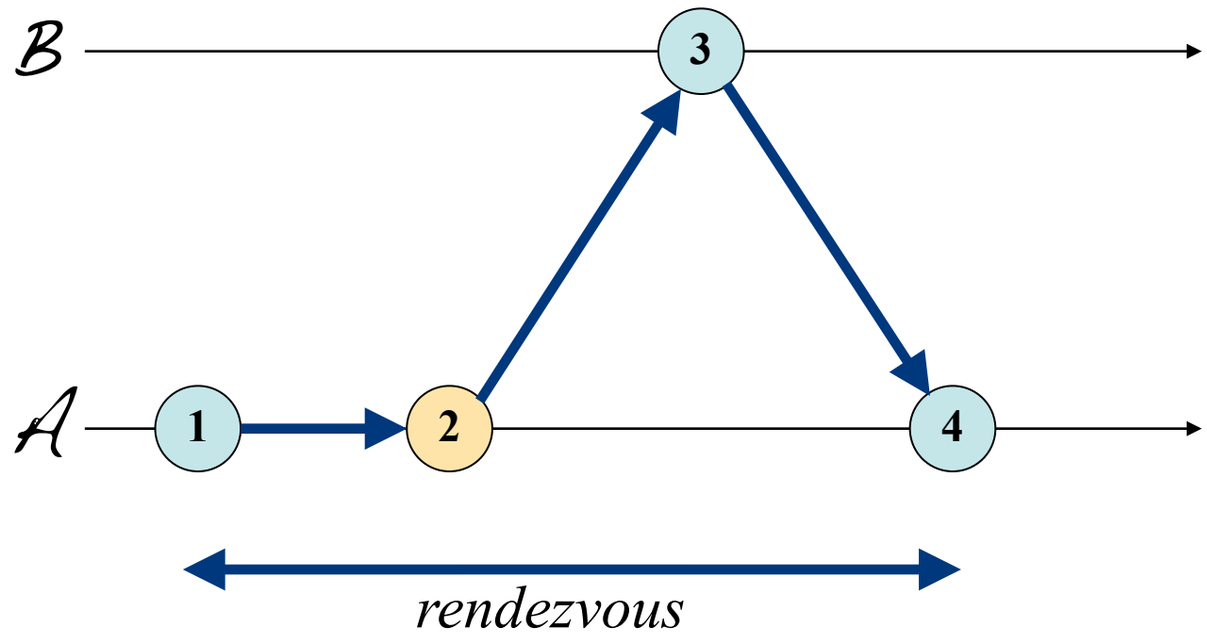
1. Request creation
2. Actual sending of the request
3. Queuing into the receiver's request queue
4. Receiving the acknowledgement



ProActive model overview
Asynchronous Sequential Processes

The rendezvous relies on four steps

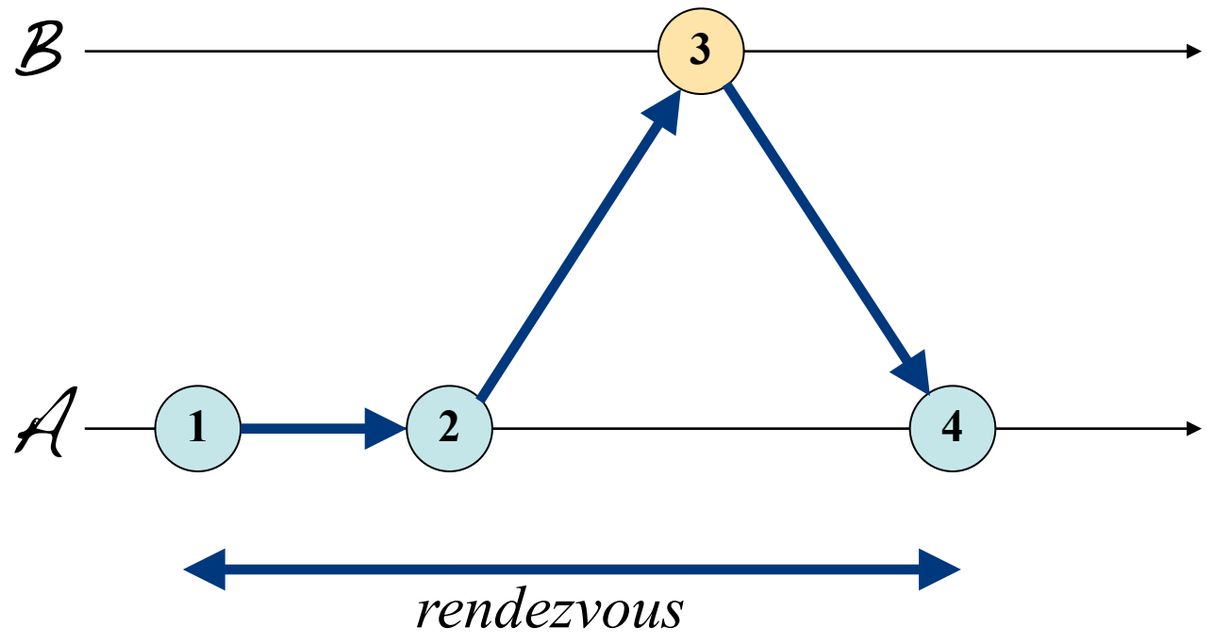
1. Request creation
2. Actual sending of the request
3. Queuing into the receiver's request queue
4. Receiving the acknowledgement



ProActive model overview
Asynchronous Sequential Processes

The rendezvous relies on four steps

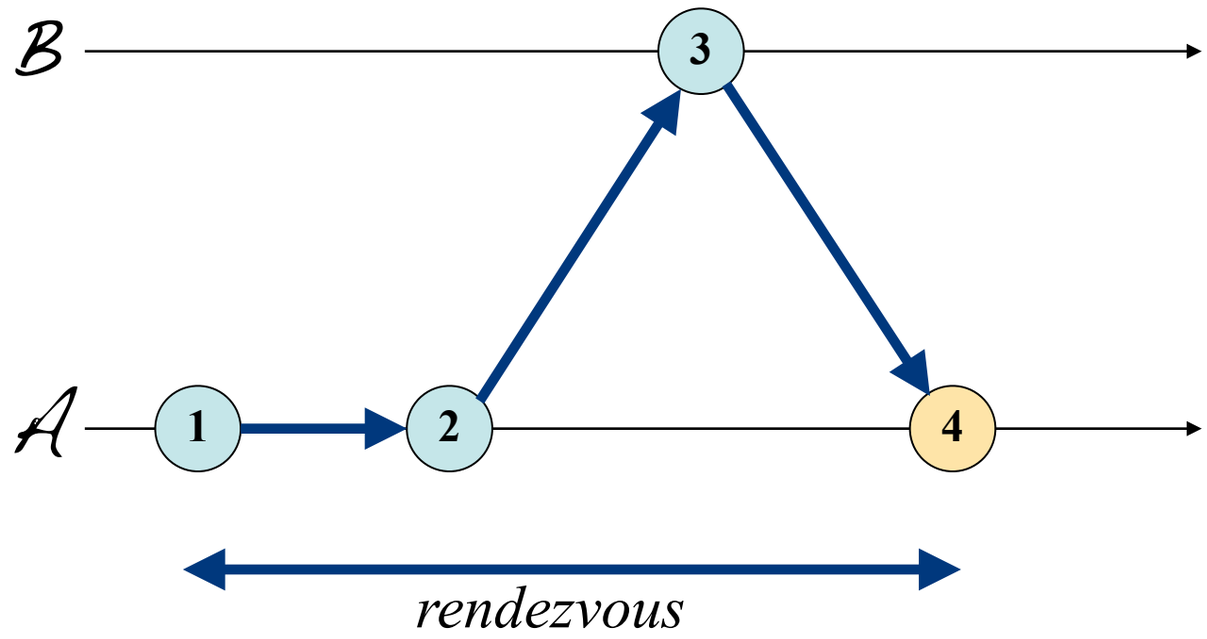
1. Request creation
2. Actual sending of the request
3. Queuing into the receiver's request queue
4. Receiving the acknowledgement



ProActive model overview
Asynchronous Sequential Processes

The rendezvous relies on four steps

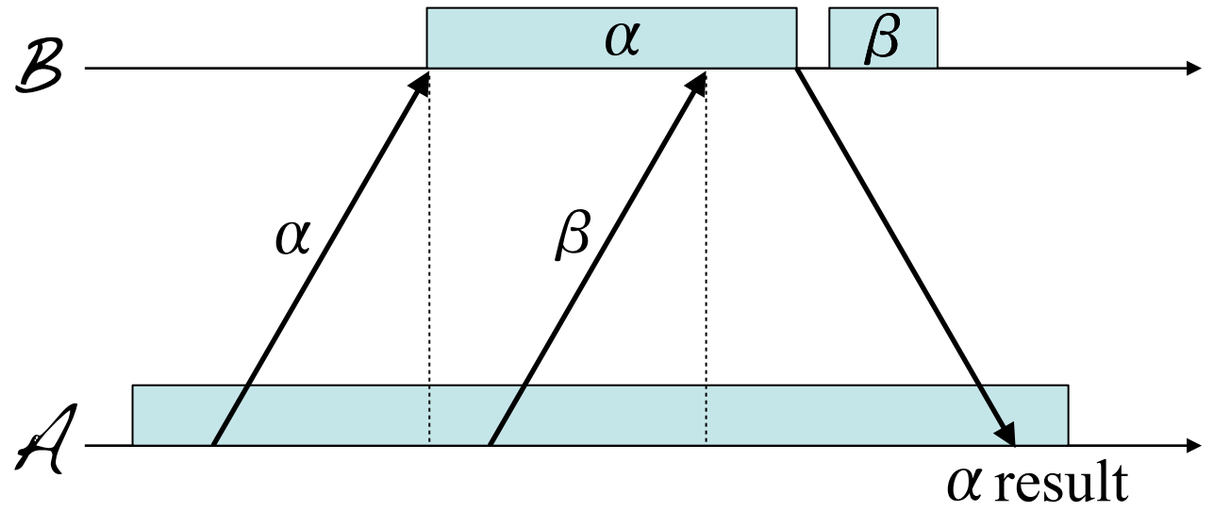
1. Request creation
2. Actual sending of the request
3. Queuing into the receiver's request queue
4. Receiving the acknowledgement



Goal : Making the rendezvous in concurrence with the computation

Two difficulties

1. Manage concurrency on the request content
2. Keep ordering between the sendings



Manage & characterize the requests

Managing the concurrency on the request content

- Common solution : *copy* or *explicit synchronization (ex: MPI_Wait)*
- Proposition : **ForgetOnSend** language construct to declare the contents which are not used after their sending anymore
- Wait-by-necessity : message-sending driven synchronization

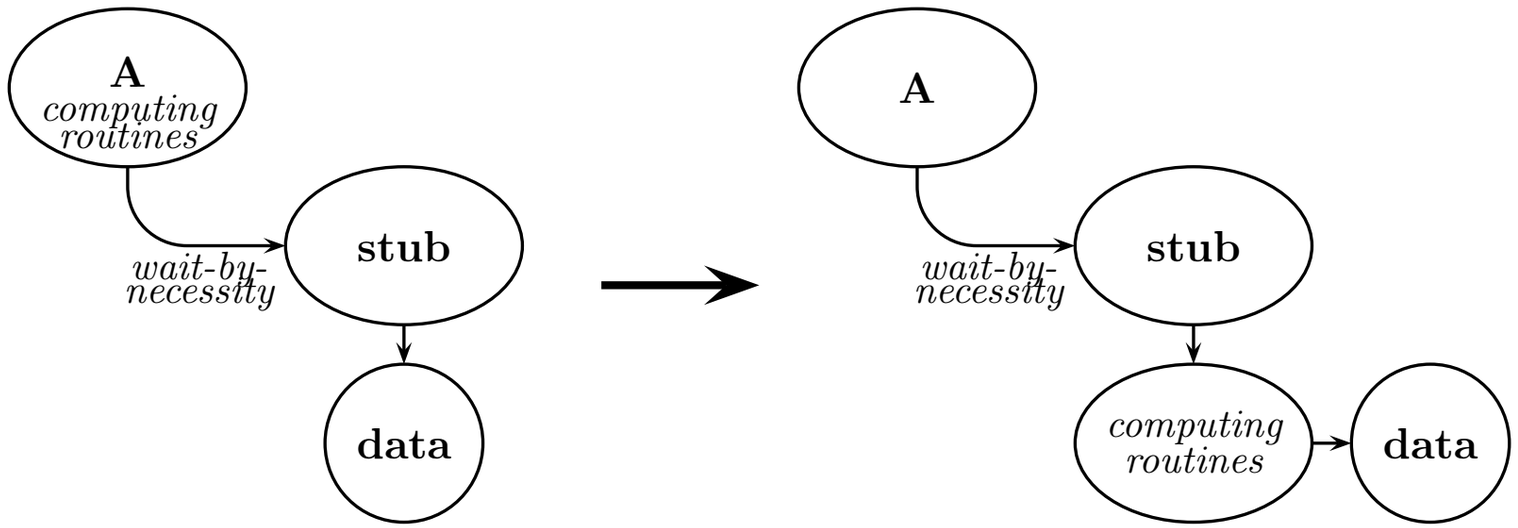
Manage & characterize the requests

Managing the concurrency on the request content

- Common solution : *copy* or *explicit synchronization (ex: MPI_Wait)*

- Proposition : **ForgetOnSend** language construct to declare the contents which are not used after their sending anymore

- Wait-by-necessity : message-sending driven synchronization



(An inefficient way) (An efficient way)

Manage & characterize the requests

Characterizing the behavior of a request

1. Functional
2. Read-only
3. Sterile

Definition (Functional Request) :
Functional requests are those which are related to the computation.

Manage & characterize the requests

Characterizing the behavior of a request

1. Functional
2. Read-only
3. Sterile

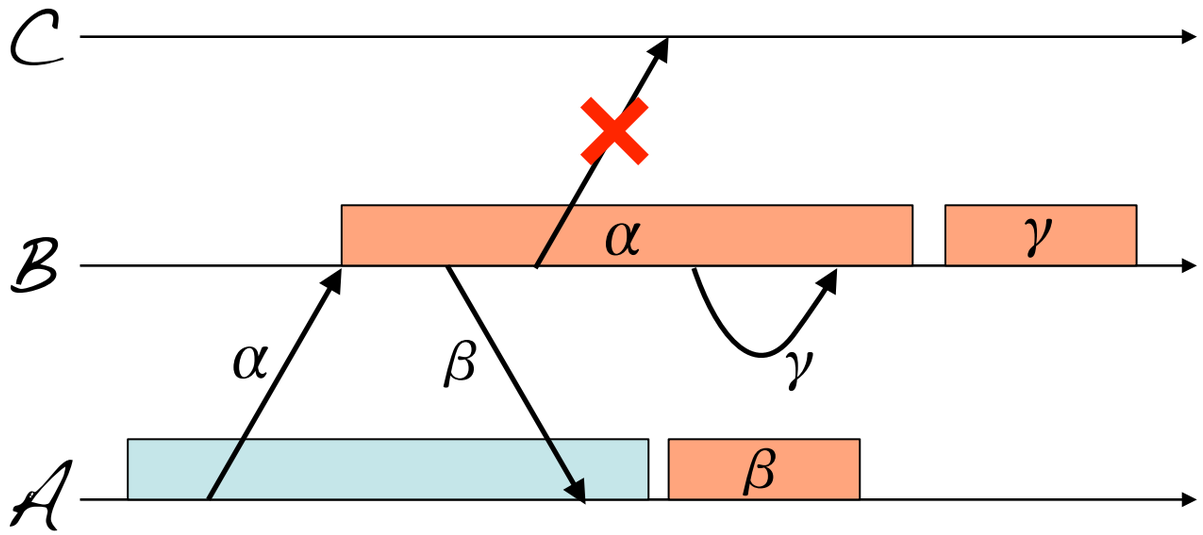
Definition (Read-only Request) :
Read-only requests are those whose the service will have no side effect on the targeted activity.

Manage & characterize the requests

Characterizing the behavior of a request

1. Functional
2. Read-only
3. Sterile

Definition (Sterile Request) :
A sterile request is a request whose the service will not imply the sending of a request, except to itself or its sender. These outgoing requests are sterile as well.



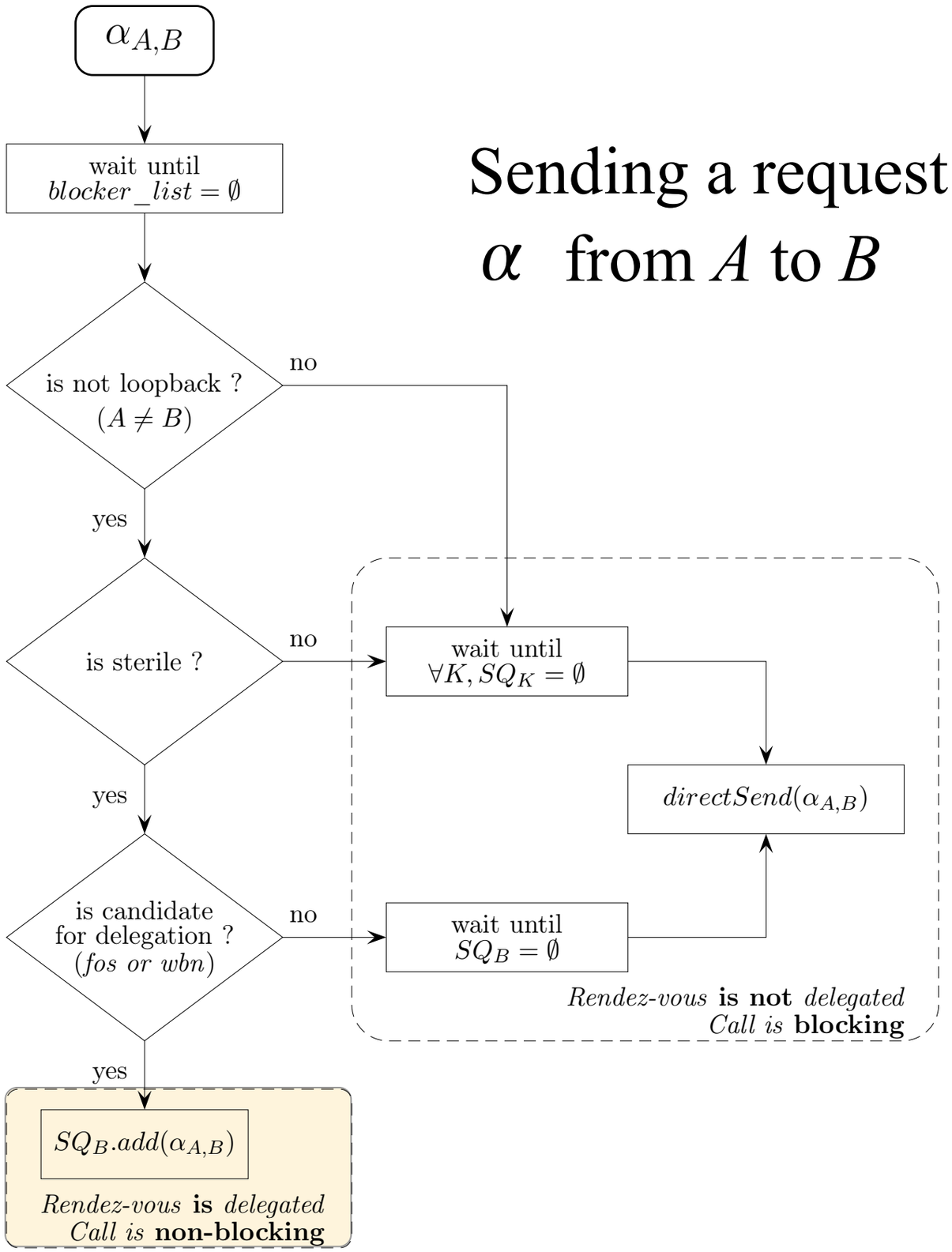
Losing rendezvous
Algorithm

- each activity has multiple sending queues (SQ_K): one per remote activity

- concurrency on the request content is handled by the ***ForgetOnSend*** contract, or the ***wait-by-necessity*** mechanism

- causal ordering is ensured with the ***sterility*** and the ***read-only*** characterizations

- incompatible requests are managed with synchronizations on the sending queues



Sending a request
α from A to B

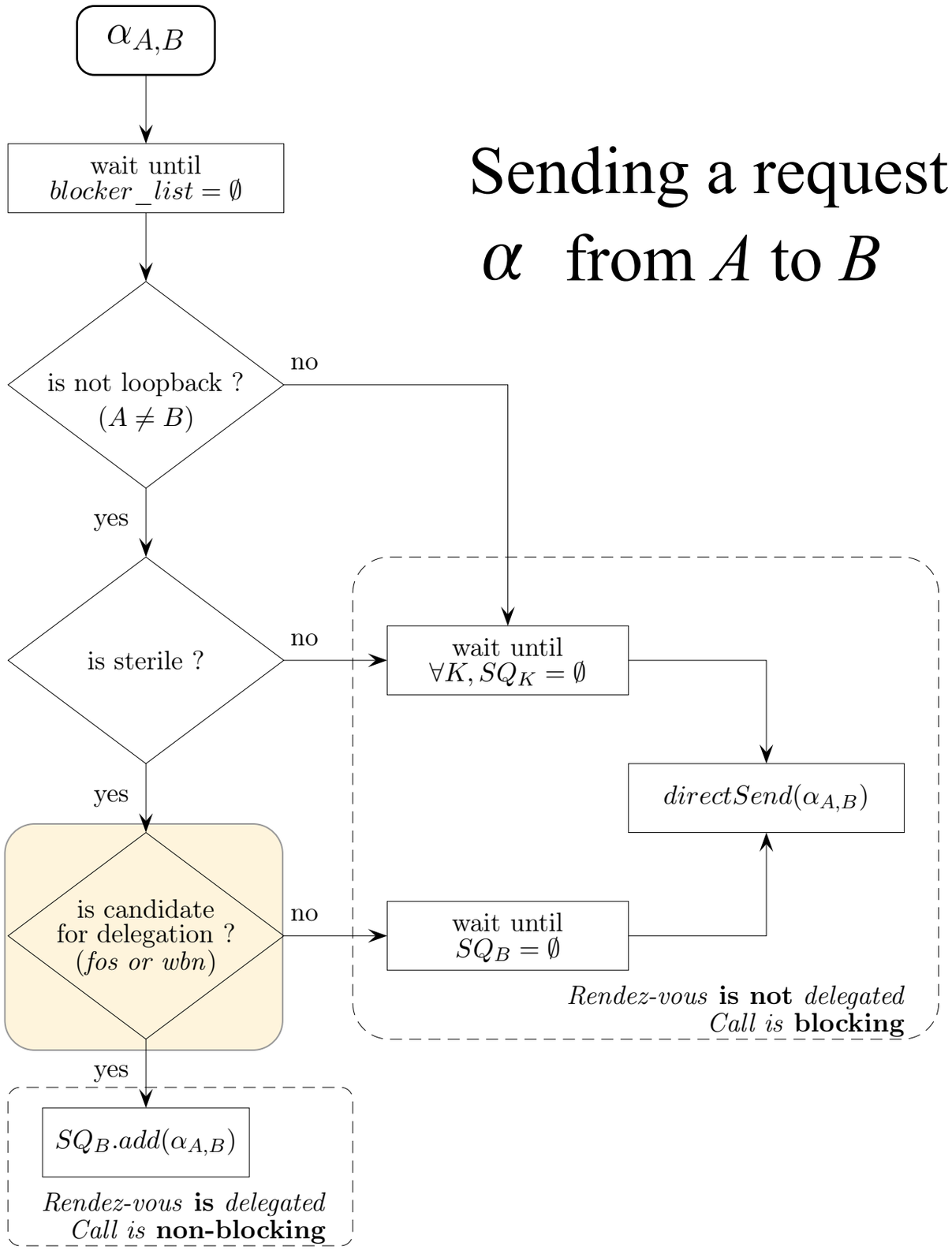
Losing rendezvous
Algorithm

- each activity has multiple sending queues (SQ_K): one per remote activity

- concurrency on the request content is handled by the ***ForgetOnSend*** contract, or the ***wait-by-necessity*** mechanism

- causal ordering is ensured with the ***sterility*** and the ***read-only*** characterizations

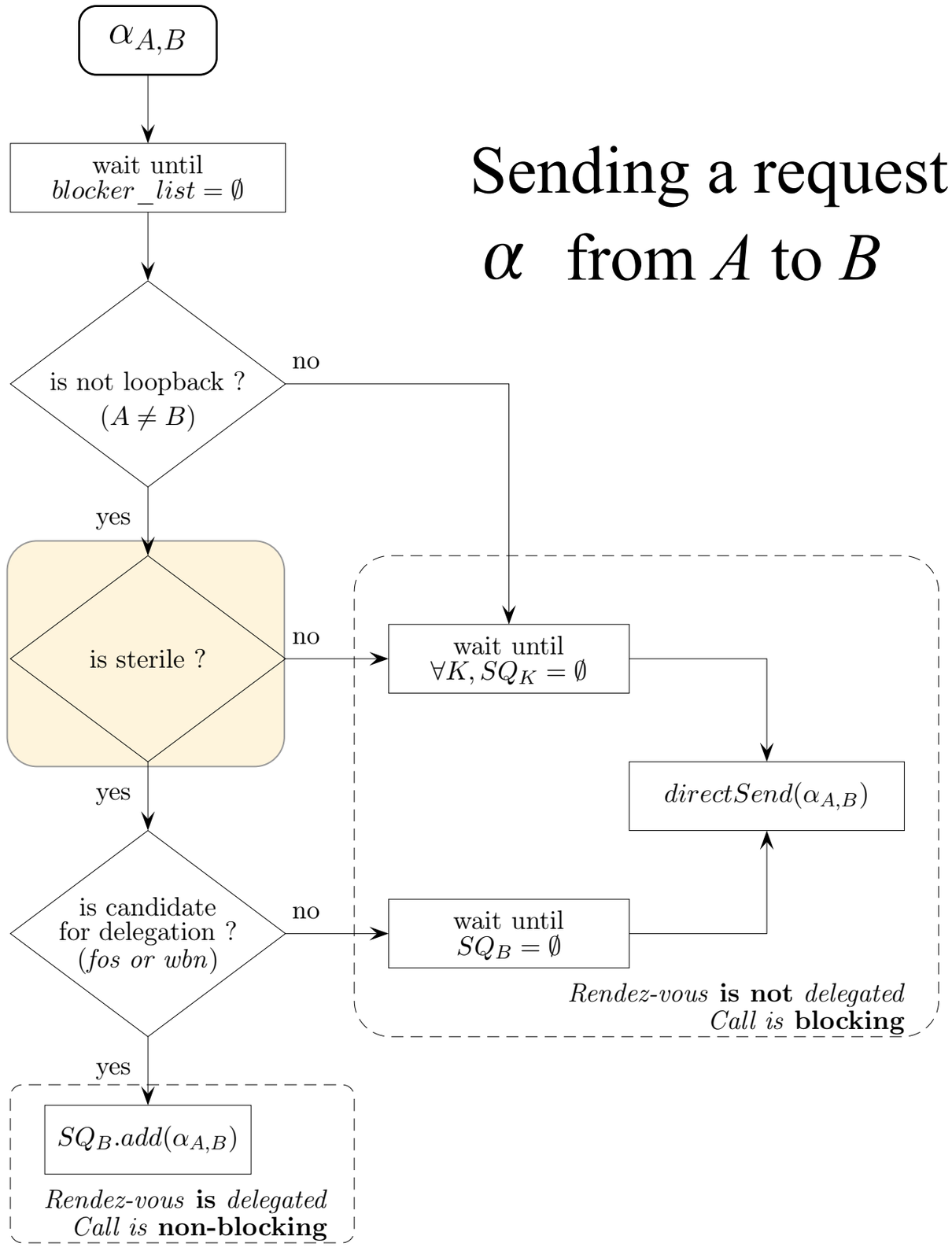
- incompatible requests are managed with synchronizations on the sending queues



Sending a request α from A to B

Losing rendezvous
Algorithm

- each activity has multiple sending queues (SQ_K): one per remote activity
- concurrency on the request content is handled by the ***ForgetOnSend*** contract, or the ***wait-by-necessity*** mechanism
- causal ordering is ensured with the ***sterility*** and the ***read-only*** characterizations
- incompatible requests are managed with synchronizations on the sending queues



Sending a request α from A to B

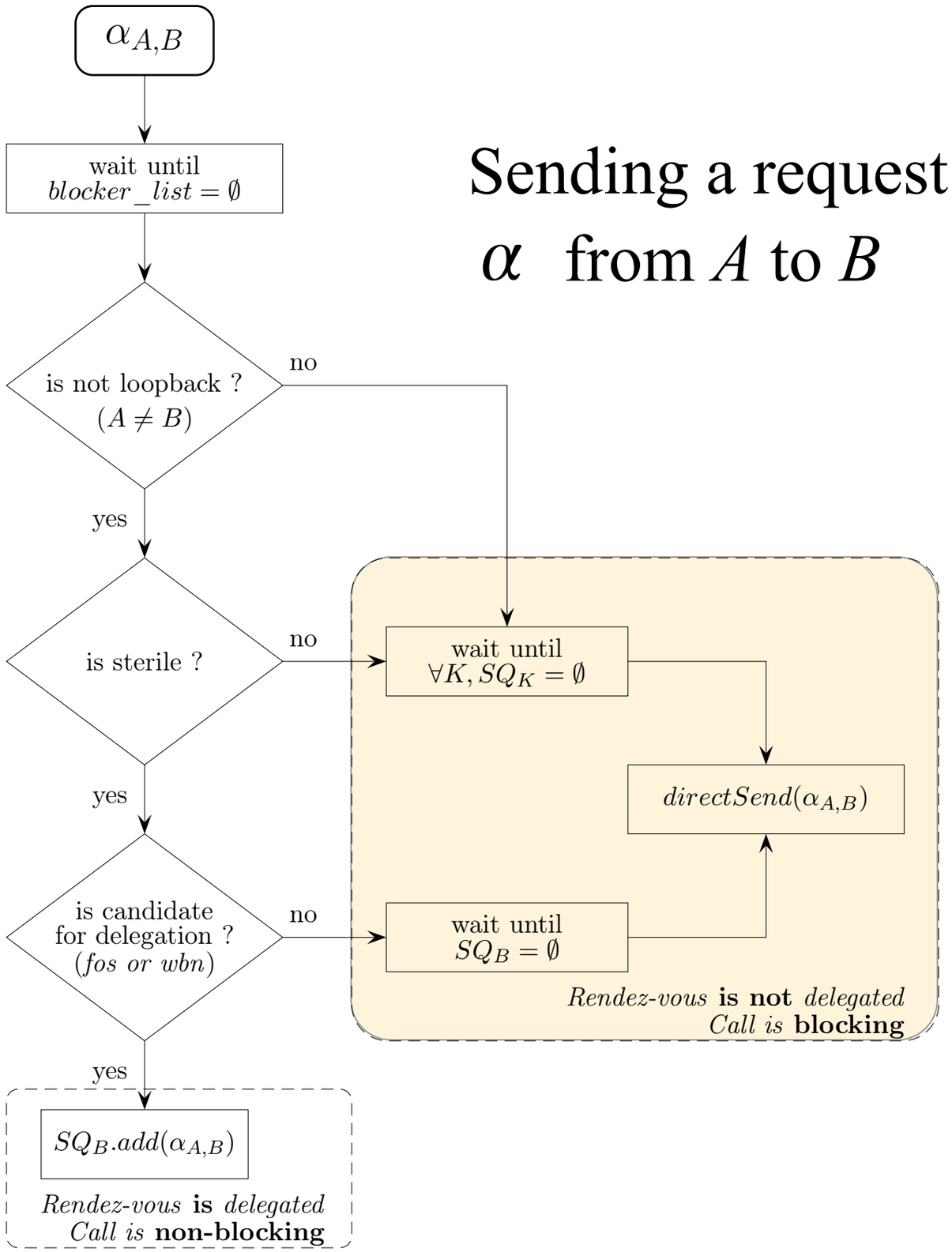
Losing rendezvous
Algorithm

- each activity has multiple sending queues (SQ_K): one per remote activity

- concurrency on the request content is handled by the ***ForgetOnSend*** contract, or the ***wait-by-necessity*** mechanism

- causal ordering is ensured with the ***sterility*** and the ***read-only*** characterizations

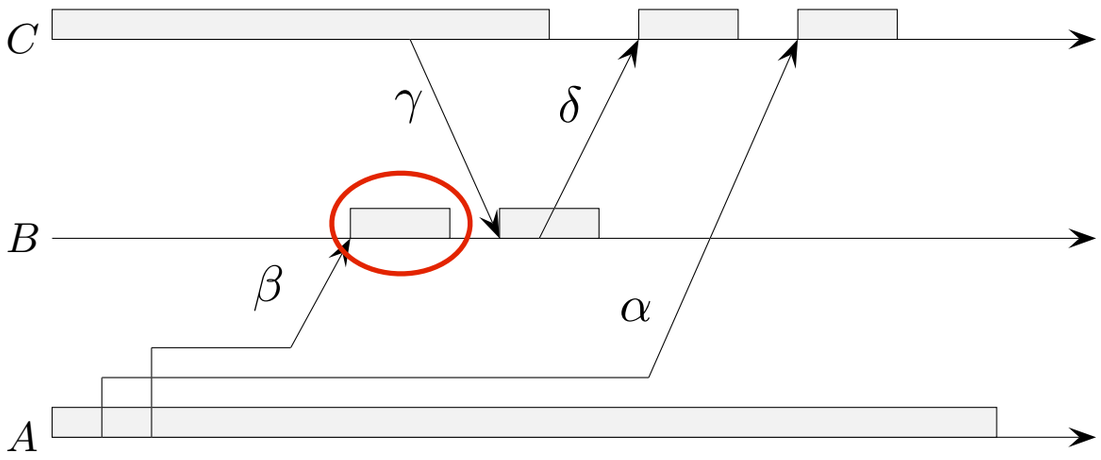
- incompatible requests are managed with synchronizations on the sending queues



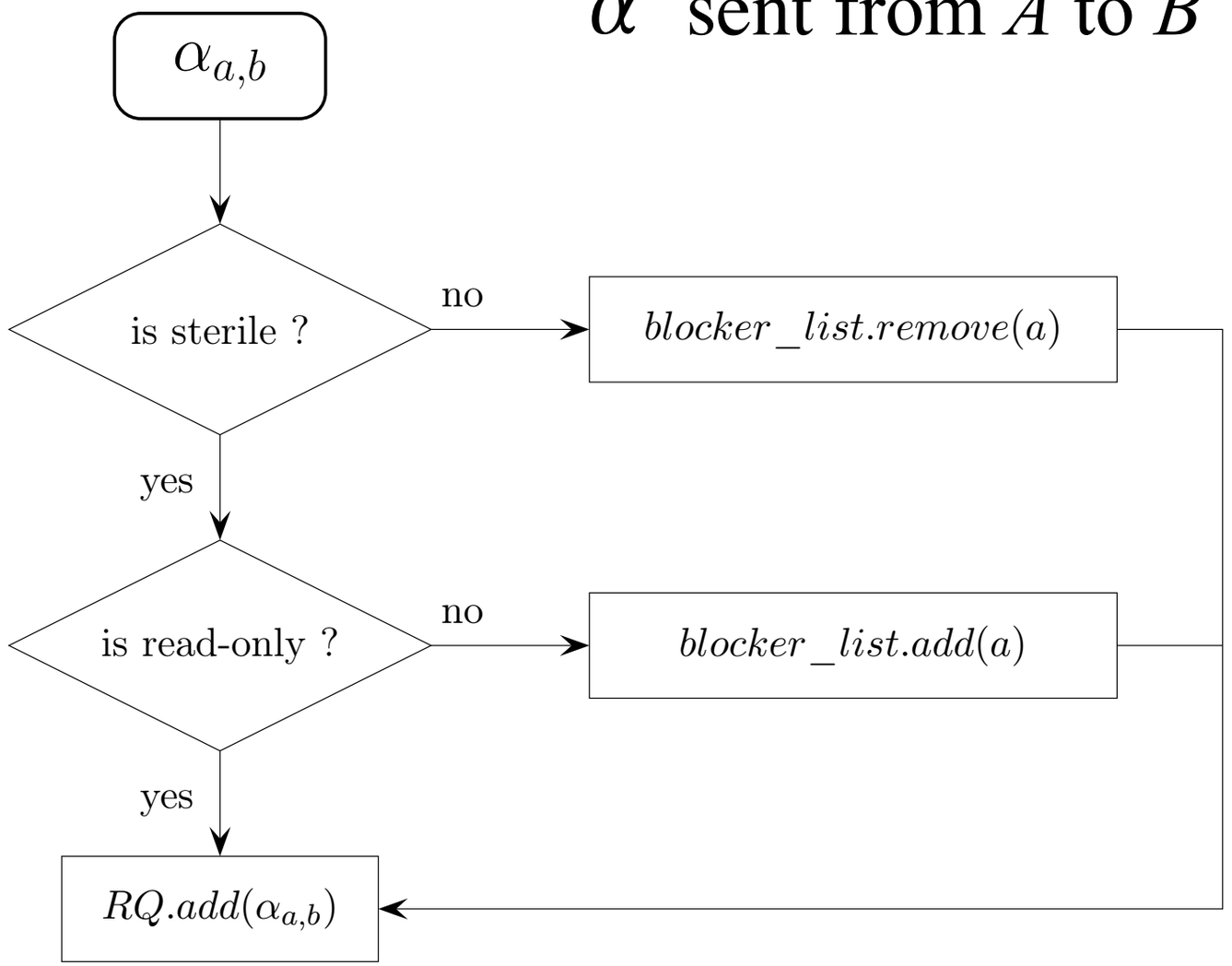
Sending a request
α from A to B

Losing rendezvous
Algorithm

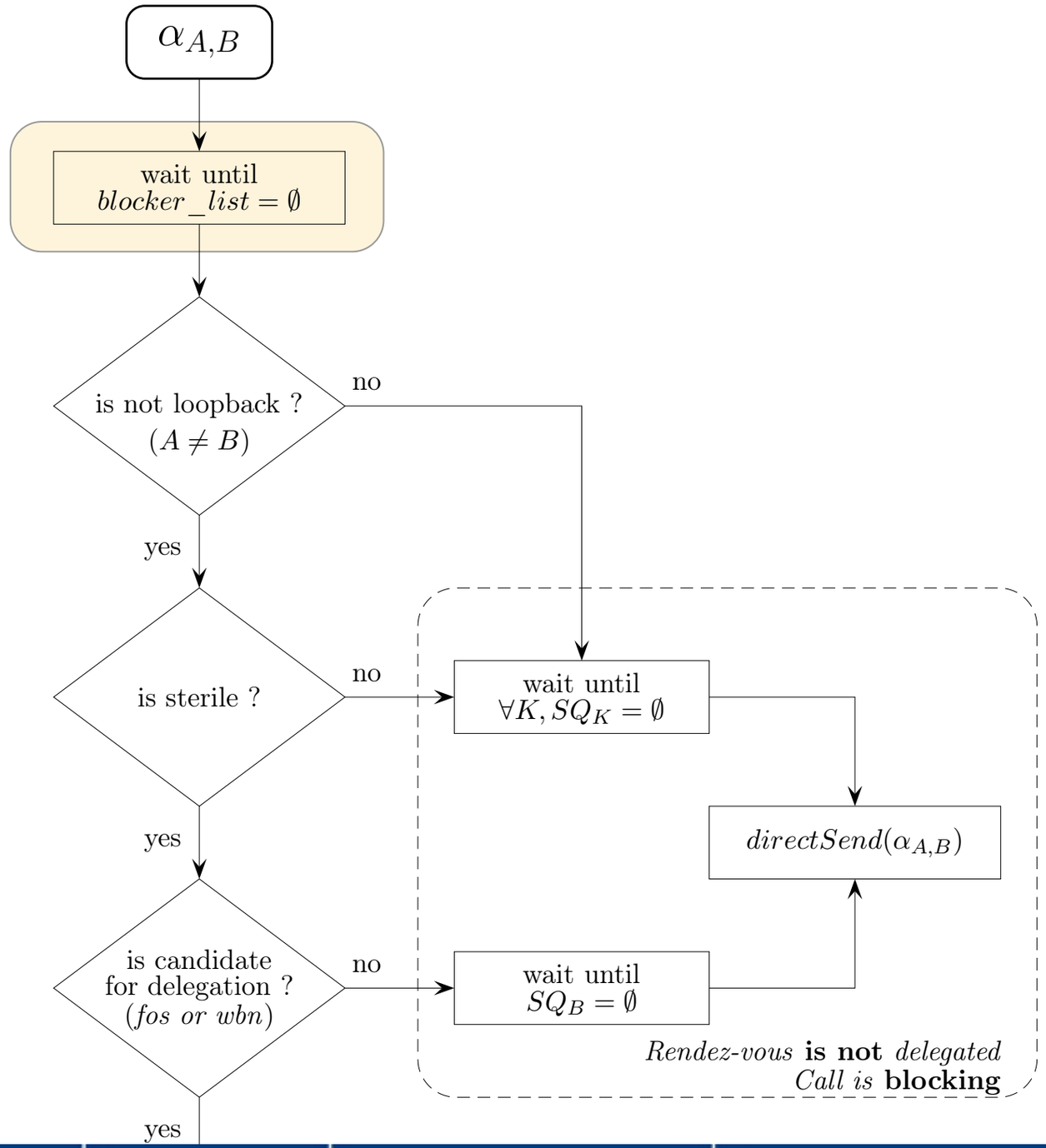
- Receiving an inappropriate request can induce a causal ordering violation



Receiving a request α sent from A to B



Losing rendezvous
Algorithm



Summary

- Managing the concurrence on the request content
  - **ForgetOnSend** language construct
  - **Wait-by-necessity** with integrated computing routines
- Characterizing the requests
  - Functional, Read-Only
  - **Sterility definition**
- Algorithms to lose the rendezvous
  - Request sending
  - Request receiving

Thank you for your attention

Questions ?

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
SOPHIA ANTIPOLIS - MÉDITERRANÉE

Code example

```
1  public class AO {
2    @Sterile
3     public void foo(double[] largeArray) {
4      // do some computation, while keeping the sterility constraint
5    }
6
7    @Sterile
8    public void bar(MyData myData) {
9      // do some computation, while keeping the sterility constraint
10 } }
```

```
1  public class MyData implements WaitByNecessityWrapper {
2    private double[] myArray;
3
4    @Override
5    public Object getData() {
6      return myArray;
7    }
8
9    public void compute() {
10     // do some computation directly on myArray
11 } }
```

```
1  public static void main(String[] args) {
2    AO ao = (AO) PAActiveObject.newActive(AO.class.getName(), null);
3    MyData mydata = (MyData) PAActiveObject.newWaitByNecessityWrapper(
4                            MyData.class.getName(), null);
5    PAActiveObject.setForgetOnSend(ao, "foo");
6    ao.foo(largeArray);
7    ao.bar(mydata);
8    ... // do some computation using neither largeArray nor mydata variables
9    md.compute();
10 }
```

Code example

```
 1  public class AO {
 2    @Sterile
 3     public void foo(double[] largeArray) {
 4      // do some computation, while keeping the sterility constraint
 5    }
 6
 7    @Sterile
 8    public void bar(MyData myData) {
 9      // do some computation, while keeping the sterility constraint
10  } }
```

```
3
4    @Override
5    public Object getData() {
6      return myArray;
7    }
8
9    public void compute() {
10     // do some computation directly on myArray
11 } }
```

```
1  public static void main(String[] args) {
2    AO ao = (AO) PAActiveObject.newActive(AO.class.getName(), null);
3    MyData mydata = (MyData) PAActiveObject.newWaitByNecessityWrapper(
4                            MyData.class.getName(), null);
5    PAActiveObject.setForgetOnSend(ao, "foo");
6    ao.foo(largeArray);
7    ao.bar(mydata);
8    ... // do some computation using neither largeArray nor mydata variables
9    md.compute();
10 }
```

Code example

```
1  public class AO {
2    @Sterile
3     public void foo(double[] largeArray) {
4      // do some computation, while keeping the sterility constraint
5    }
6
7    @Sterile
8    public void bar(MyData myData) {
9      // do some computation, while keeping the sterility constraint
```

```
1  public class MyData implements WaitByNecessityWrapper {
2    private double[] myArray;
3
4    @Override
5    public Object getData() {
6      return myArray;
7    }
8
9    public void compute() {
10     // do some computation directly on myArray
11 } }
```

```
2    AO ao = (AO) PAActiveObject.newActive(AO.class.getName(), null);
3    MyData mydata = (MyData) PAActiveObject.newWaitByNecessityWrapper(
4                            MyData.class.getName(), null);
5    PAActiveObject.setForgetOnSend(ao, "foo");
6    ao.foo(largeArray);
7    ao.bar(mydata);
8    ... // do some computation using neither largeArray nor mydata variables
9    md.compute();
10 }
```

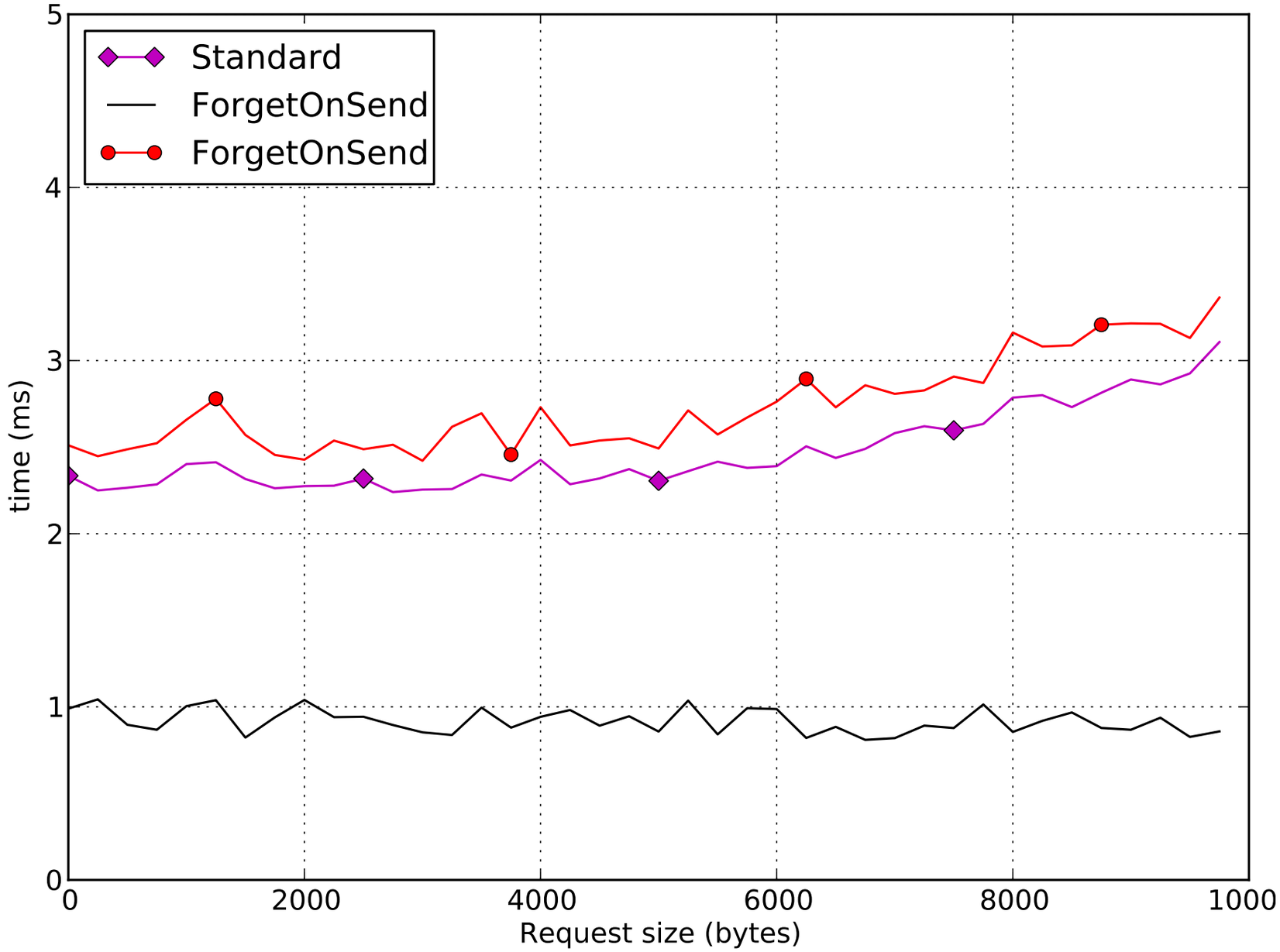
Code example

```
1  public class AO {
2    @Sterile
3     public void foo(double[] largeArray) {
4      // do some computation, while keeping the sterility constraint
5    }
6
7    @Sterile
8    public void bar(MyData myData) {
9      // do some computation, while keeping the sterility constraint
10 } }
```

```
1  public class MyData implements WaitByNecessityWrapper {
2    private double[] myArray;
3
4    @Override
5    public Object getData() {
6      return myArray;
7    }
8
9    public void compute() {
```

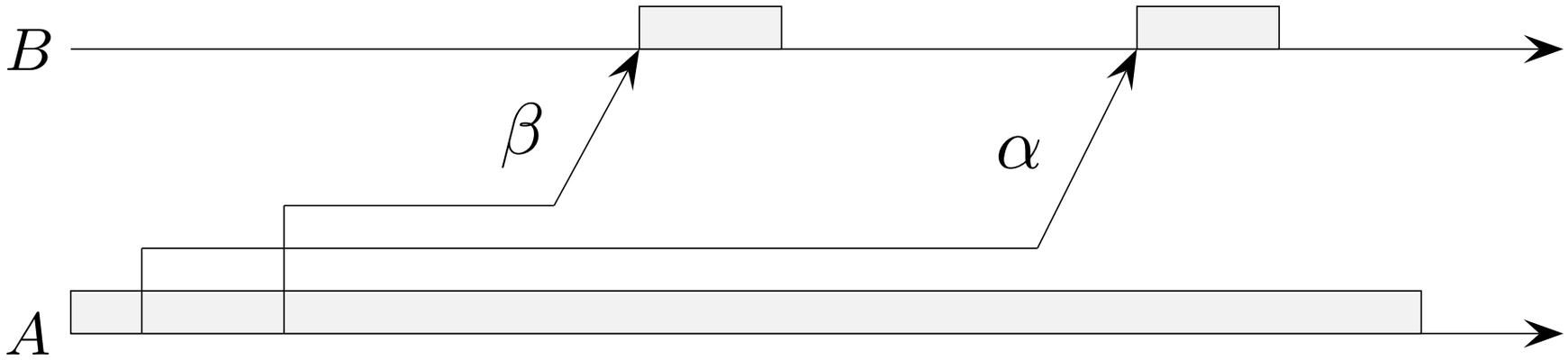
```
1  public static void main(String[] args) {
2    AO ao = (AO) PAActiveObject.newActive(AO.class.getName(), null);
3    MyData mydata = (MyData) PAActiveObject.newWaitByNecessityWrapper(
4                             MyData.class.getName(), null);
5    PAActiveObject.setForgetOnSend(ao, "foo");
6    ao.foo(largeArray);
7    ao.bar(mydata);
8    ... // do some computation using neither largeArray nor mydata variables
9    md.compute();
10 }
```

Performance



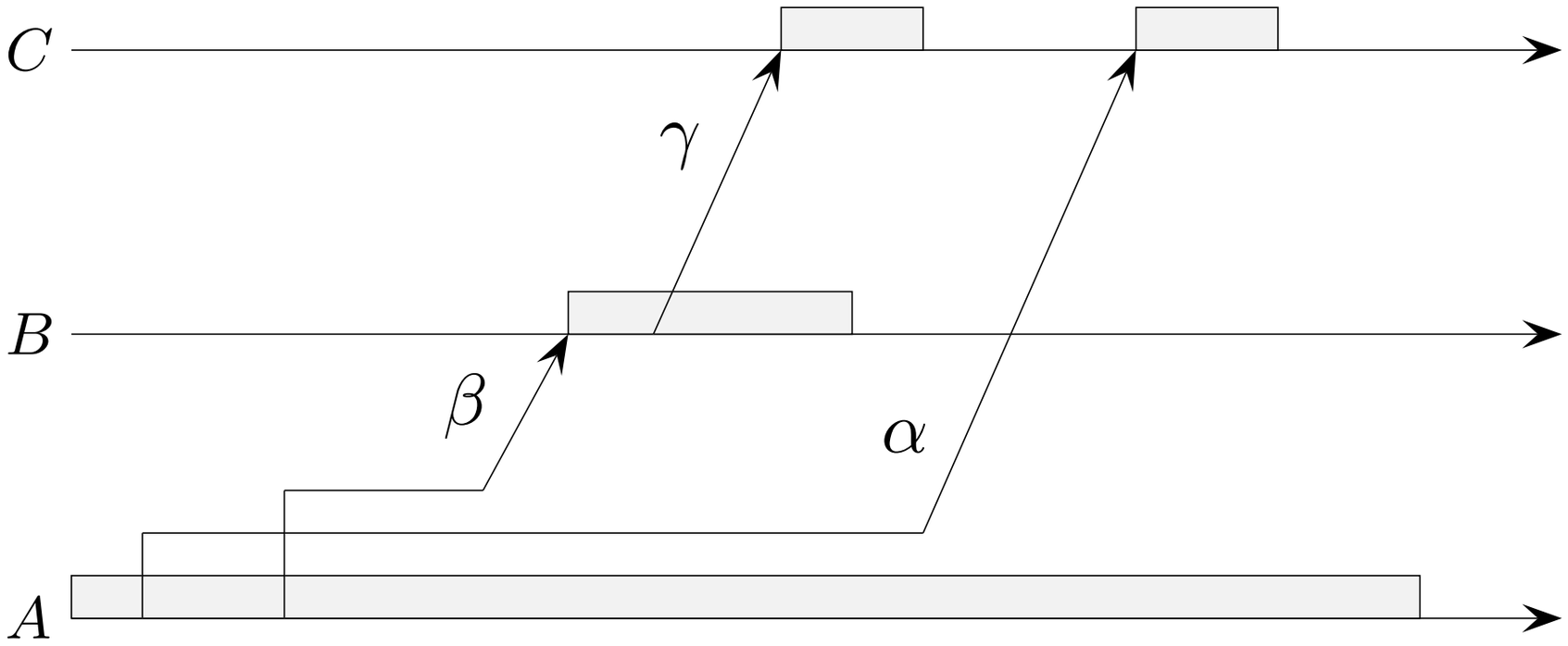
Examples of CO violations

1 - Loss of point-to-point FIFO ordering



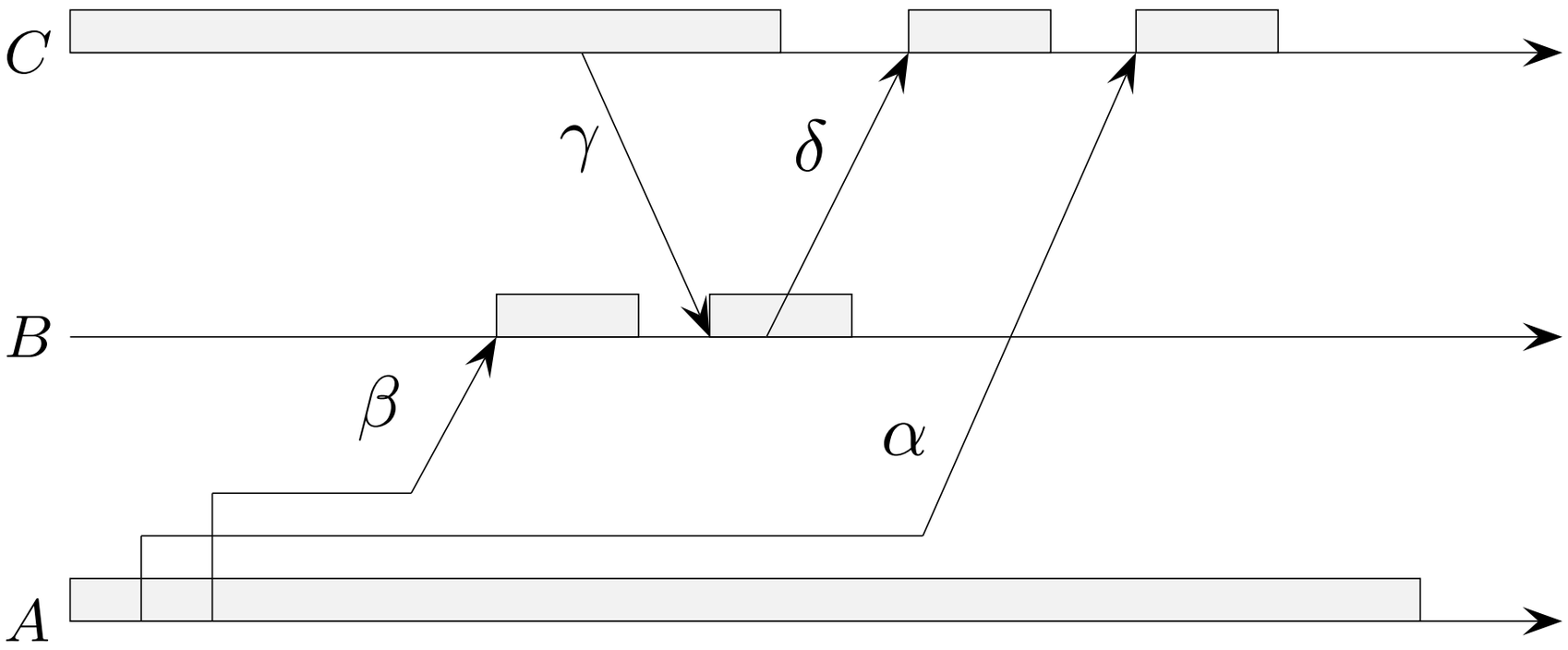
Examples of CO violations

2 - Direct loss of CO

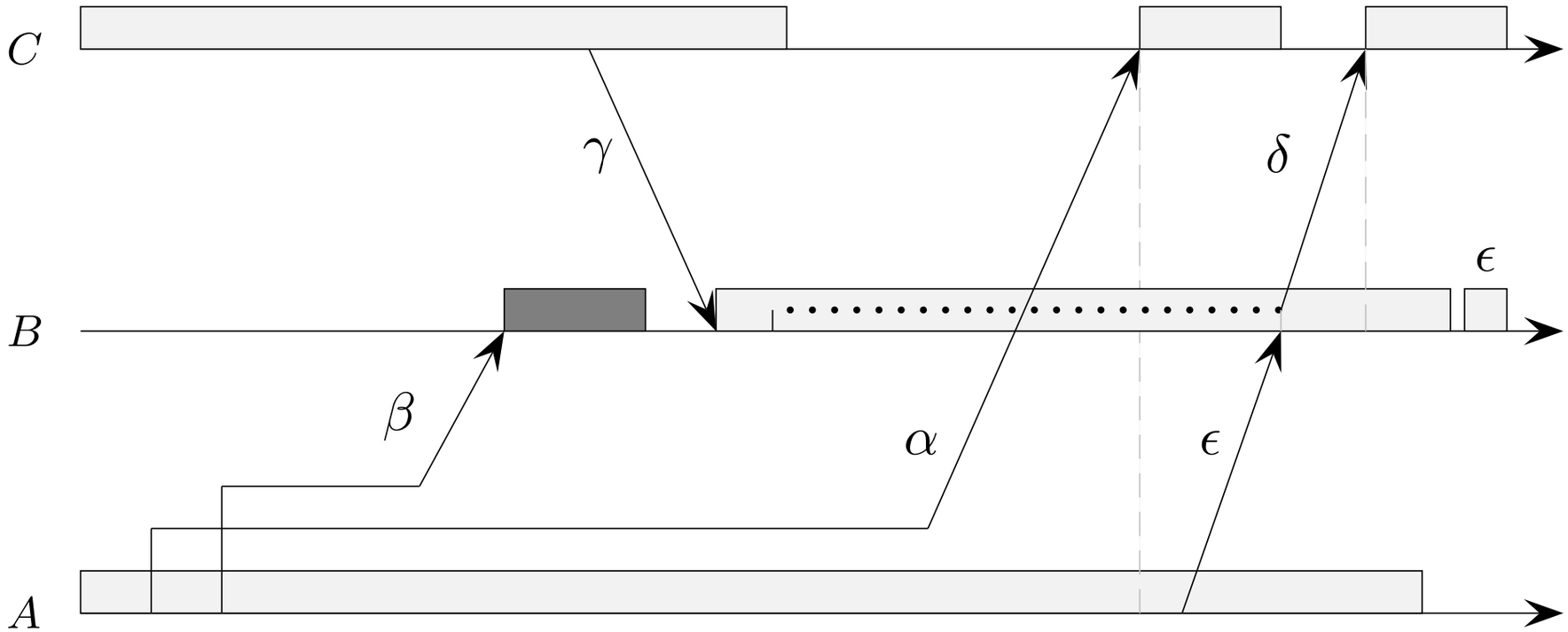


Examples of CO violations

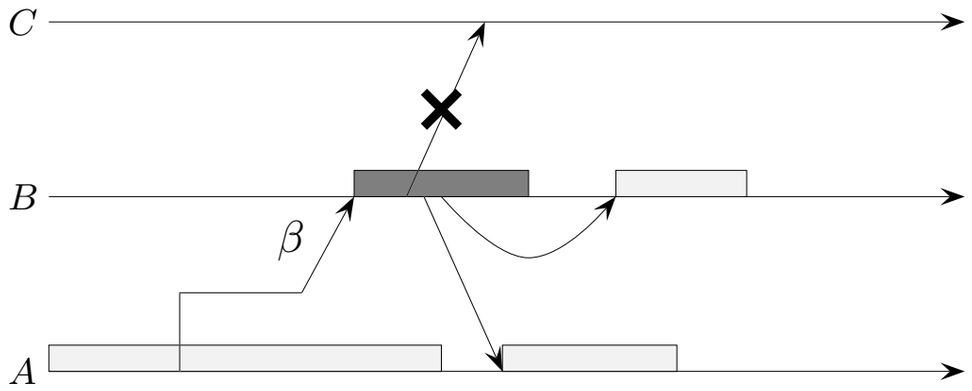
3 - Indirect loss of CO

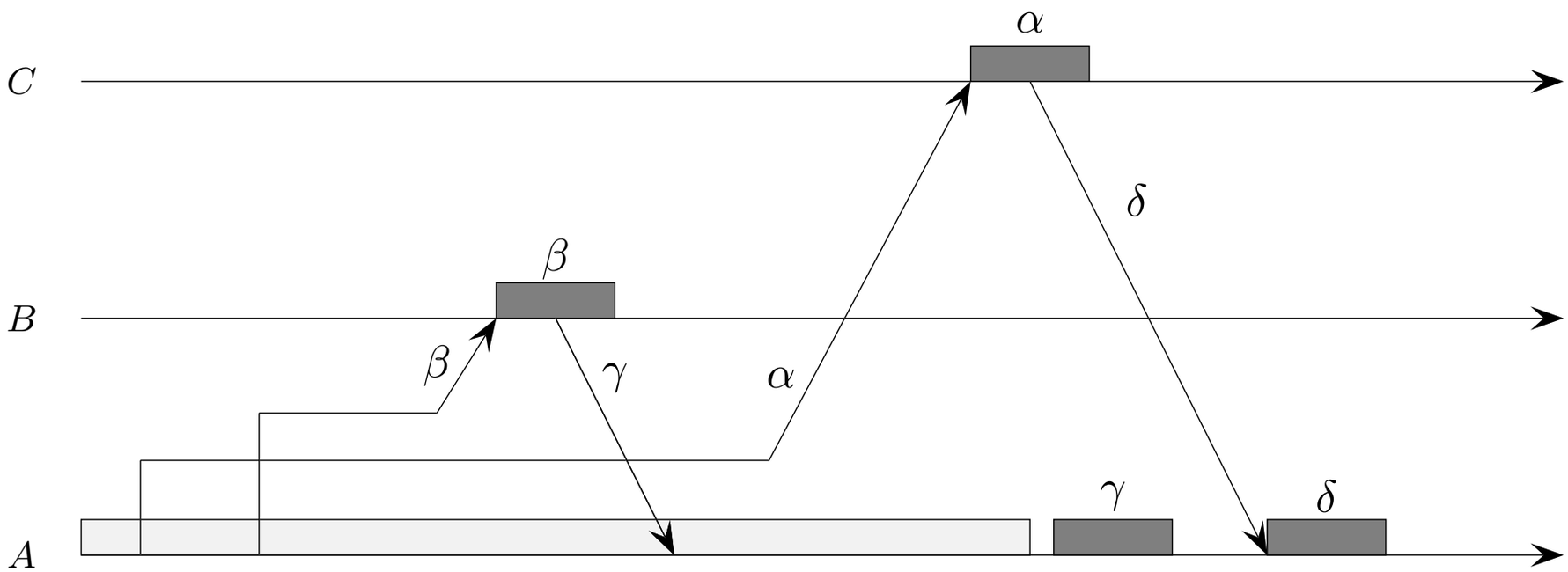


Blocking the sendings of an activity



More on the sterility definition

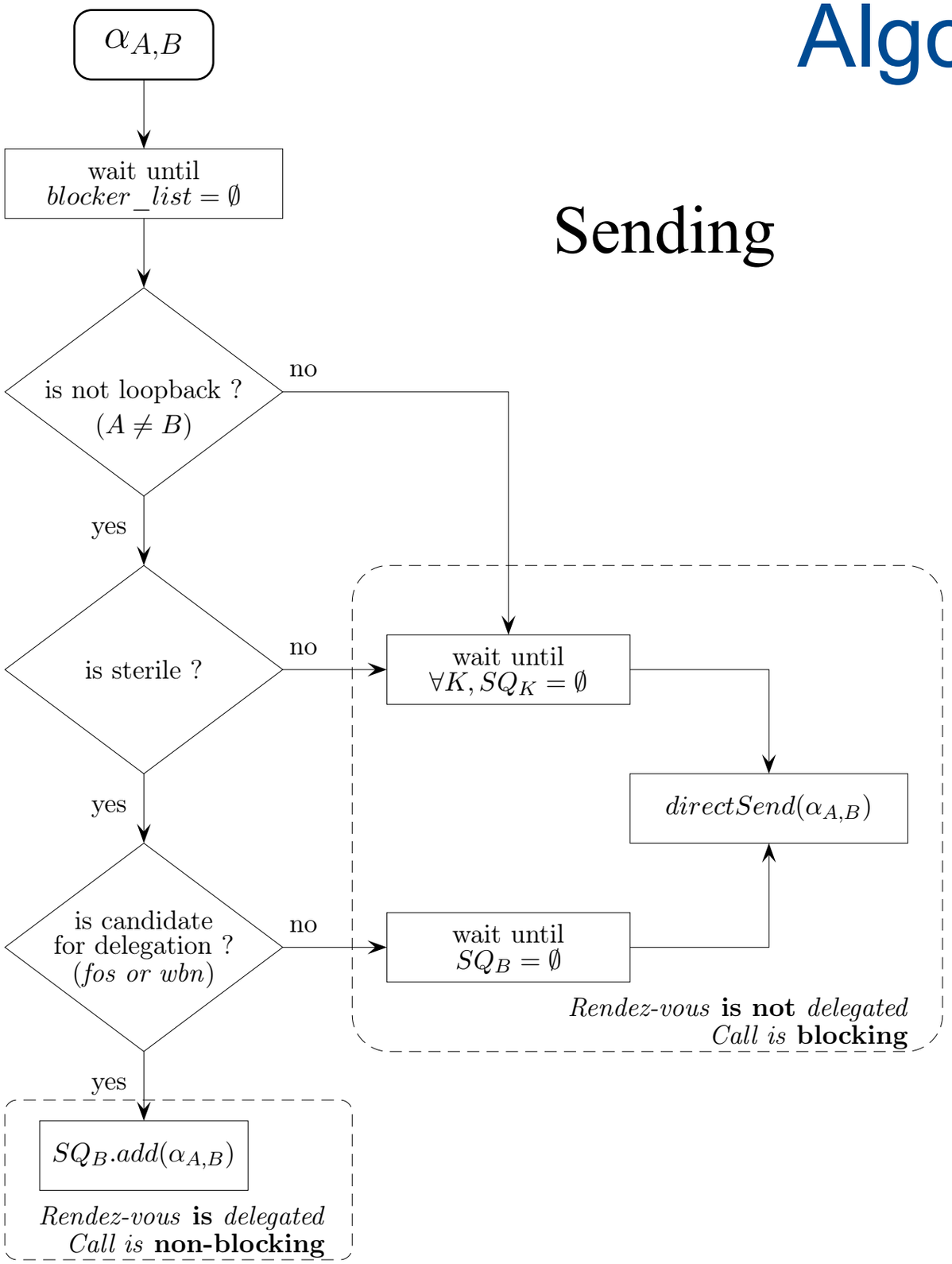




Permitting the activity to send back a sterile request to its parent cannot induces a causal ordering disruption as well

Algorithms

Sending



Receiving