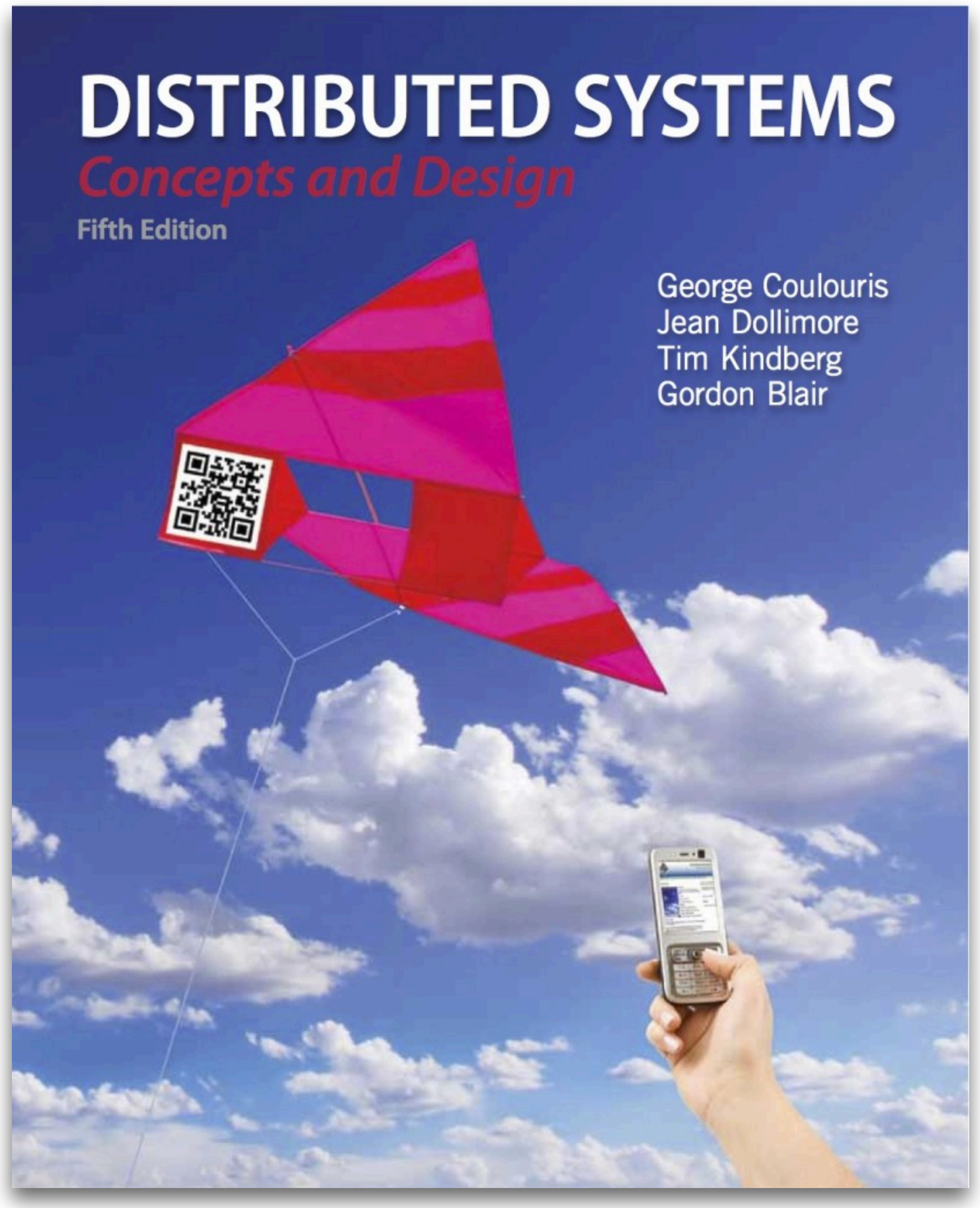
Distributed Systems 2023-2024: Indirect communication

Wouter Joosen & Tom Van Cutsem
DistriNet KU Leuven
September 2023

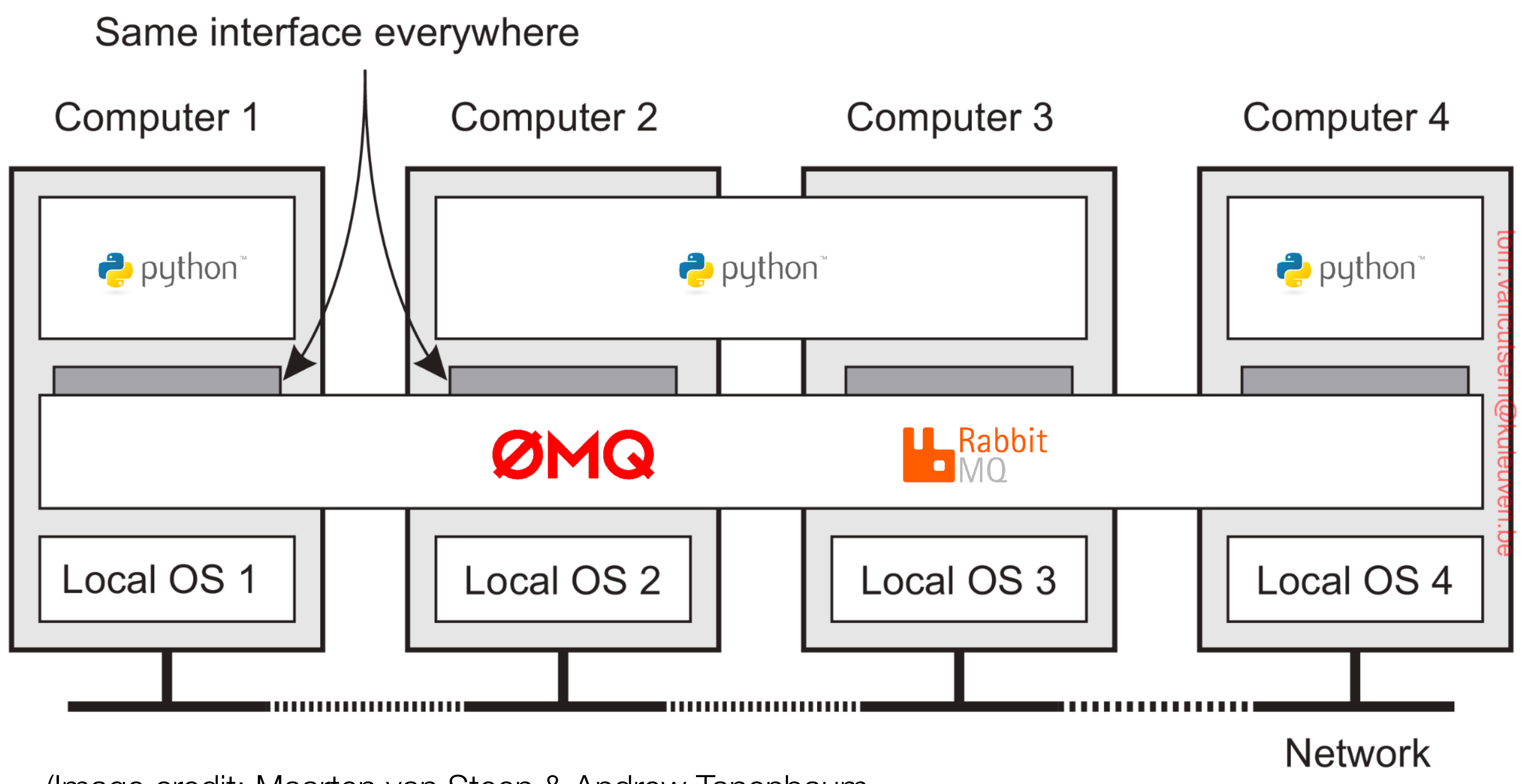
Learning resources

- Textbook, chapter 6
  - Introduction
  - (Group Communication) (revisit in later lecture on coordination)
  - Publish-subscribe systems (not 6.3.2)
  - Message queues (not 6.4.2, 6.4.3)
  - Shared memory approaches (not 6.5.1, not JavaSpaces)
  - Summary

- **Note:** we will replace the Java-oriented case studies in the book with contemporary middleware and example Python code snippets.
- See code examples with explanation on Github:
- https://github.com/tvcutsem/distributed-systems



Middleware for communication: case studies



(Image credit: Maarten van Steen & Andrew Tanenbaum, "Distributed Systems", 4th edition)

tom.vancutsem@kuleuven.be

- **ZeroMQ**: provides flexible "socket-like" API to exchange messages between processes (direct communication)
- **RabbitMQ**: a complete "message broker" based on the AMQP protocol. Supports pub-sub and message queues (indirect communication)
- Note: the details of these systems are *not* important. We will focus on the *communication patterns* and the *programming models* (APIs) that they offer to applications.

Communication Paradigms: recap from introduction

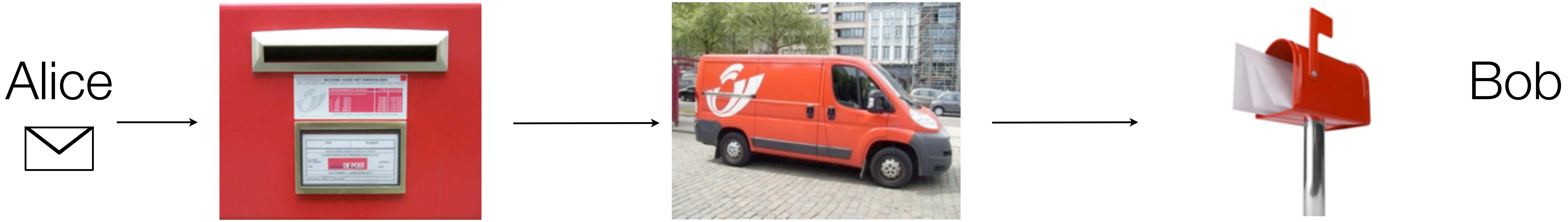
- Three major paradigms:
  - 1. **Inter-process communication** (reading/writing a byte stream shared between two processes)
    - Usually based on OS network programming APIs (e.g. UNIX sockets)
  - 2. **Remote invocation** (basic request/response interaction between two processes)
    - Remote procedure call (in procedural languages) or remote method invocation (in object-oriented languages)
  - 3. **Indirect Communication** (communication with a known or unknown group of processes)
    - Often by registering “Observers” or “Listeners” or “Callbacks” on an event source

Space and time coupling in distributed systems

- Inter-process communication (IPC) and remote invocation (RPC) communication strongly **couple** sender and receiver:
  - in **time**: sender and receiver must both be online at the same time
  - in **space**: sender must know the receiver
- This is not always acceptable

Decoupling in Time

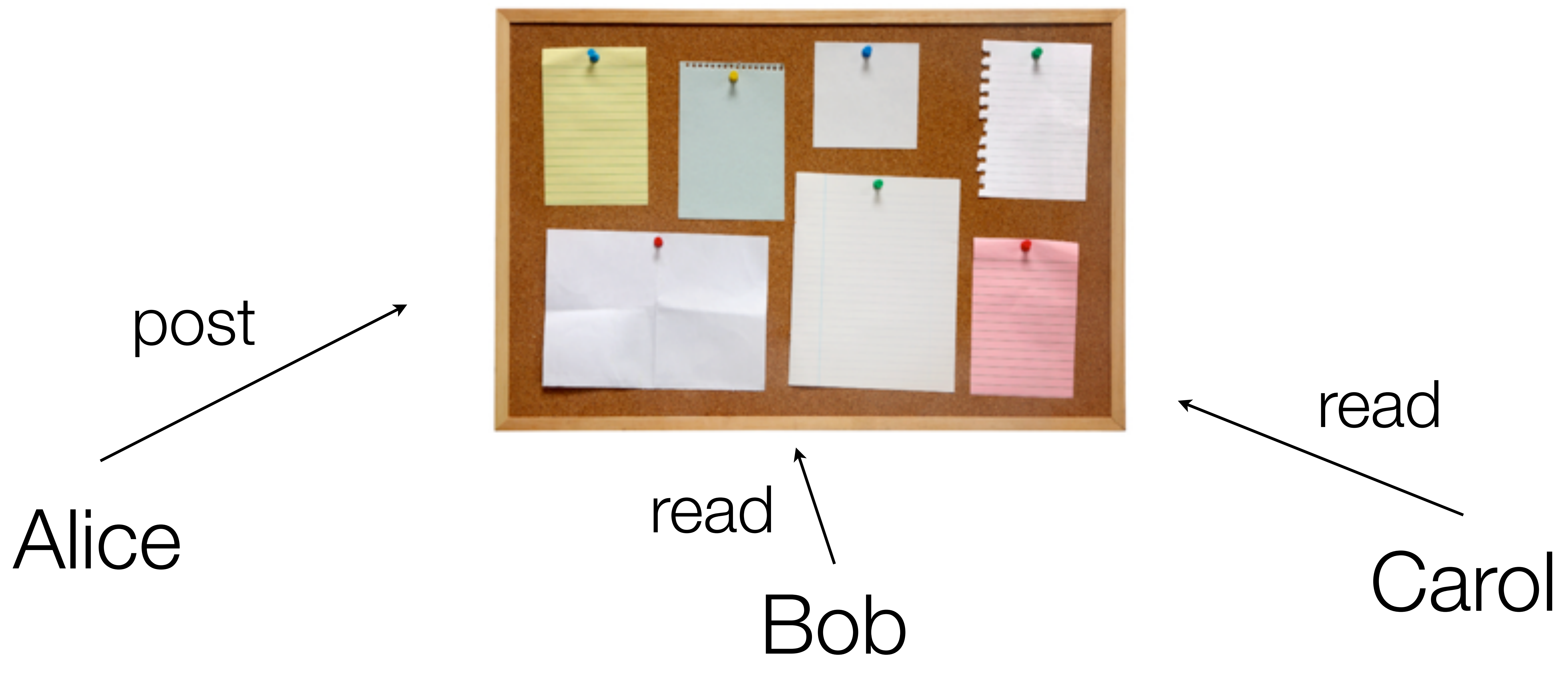
- Communicating parties **need not be online at the same time**
- Requires intermediary third-party to store communicated messages
- Example: mailboxes in postal mail and e-mail



(Alice doesn’t need to meet up with Bob to exchange her letter)

Decoupling in Space

- Sender **does not need to know the address or identity** of the receiver(s)
- Example: communication via a public bulletin-board, mailing list, etc.



(Alice doesn't need to know Bob or Carol to share her note with them)

Communication Patterns

- Request-reply
- Event Notification
- Pipeline (push-pull)

Direct communication (time and space-coupled)

- Publish-subscribe
- Message queue
- Tuple space

Indirect communication (time and space de-coupled)

Communication Patterns

- **Request-reply**
- Event Notification
- Pipeline (push-pull)

$\left.\right\}$ Direct communication (time and space-coupled)

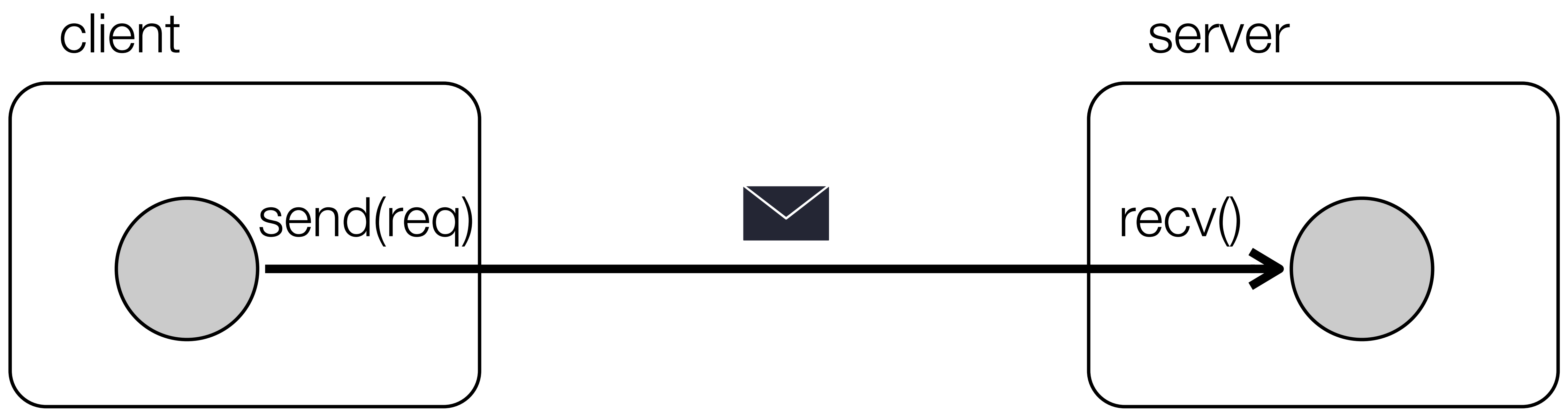
- Publish-subscribe
- Message queue
- Tuple space

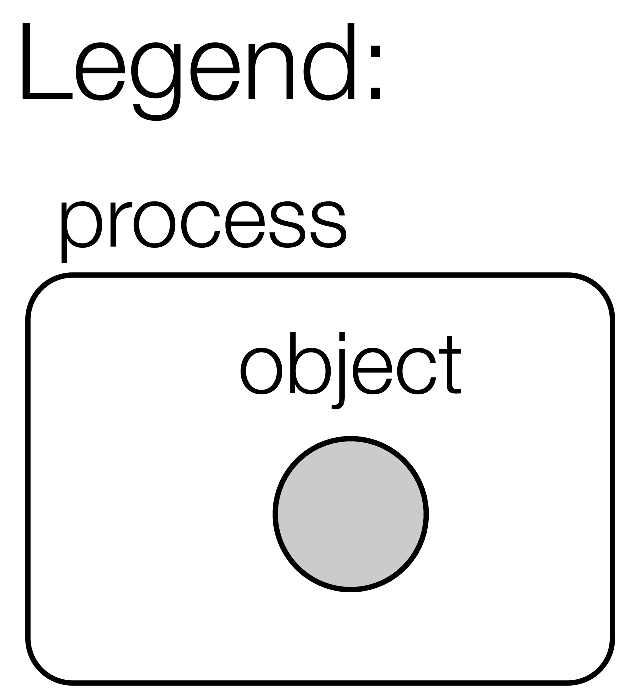
$\left.\right\}$ Indirect communication (time and space de-coupled)

KU LEUVEN DistriNet

Request-reply

- Client directly connects to server, server accepts requests from any client
- Server sends response, client blocks until response is received



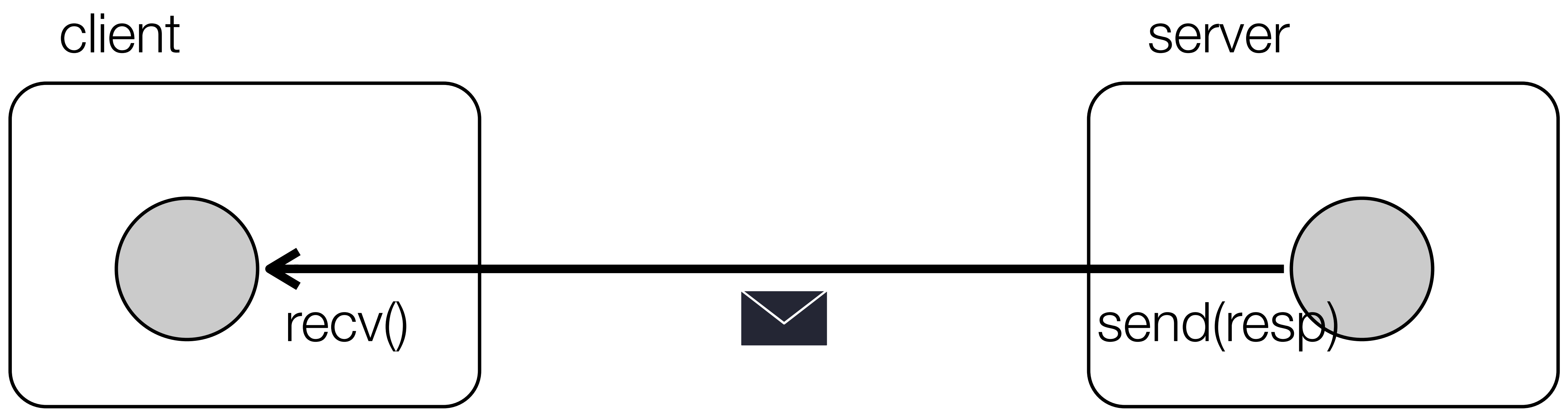


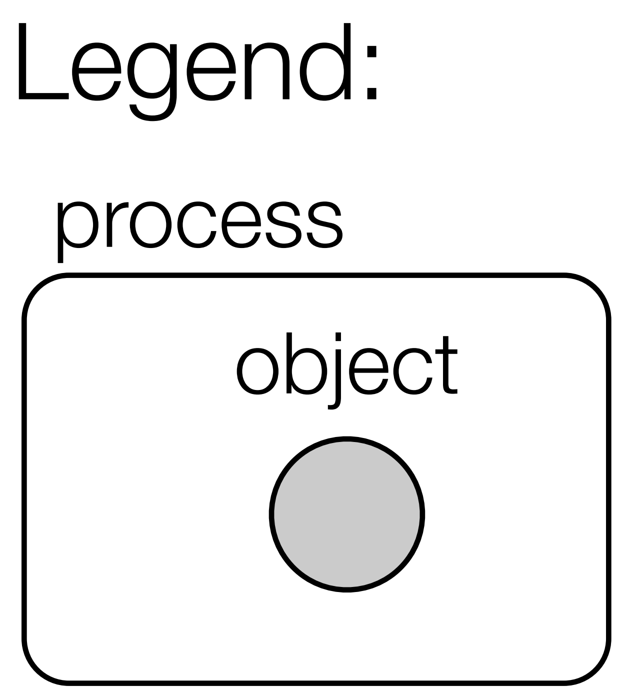
ØMQ

/request_reply

Request-reply

- Client directly connects to server, server accepts requests from any client
- Server sends response, client blocks until response is received





ØMQ

/request_reply

KU LEUVEN DistriNet

Request-reply: summary

- Time-coupled: client and server must be online when the request and the response is made.
- Space-coupled: client connects directly to known server
- Supports one-to-one communication
- Request-reply is often synchronous (as in standard remote procedure call) but can also be asynchronous (the client is free to do other work until it needs the reply)

Communication Patterns

- Request-reply
- **Event Notification**
- Pipeline (push-pull)

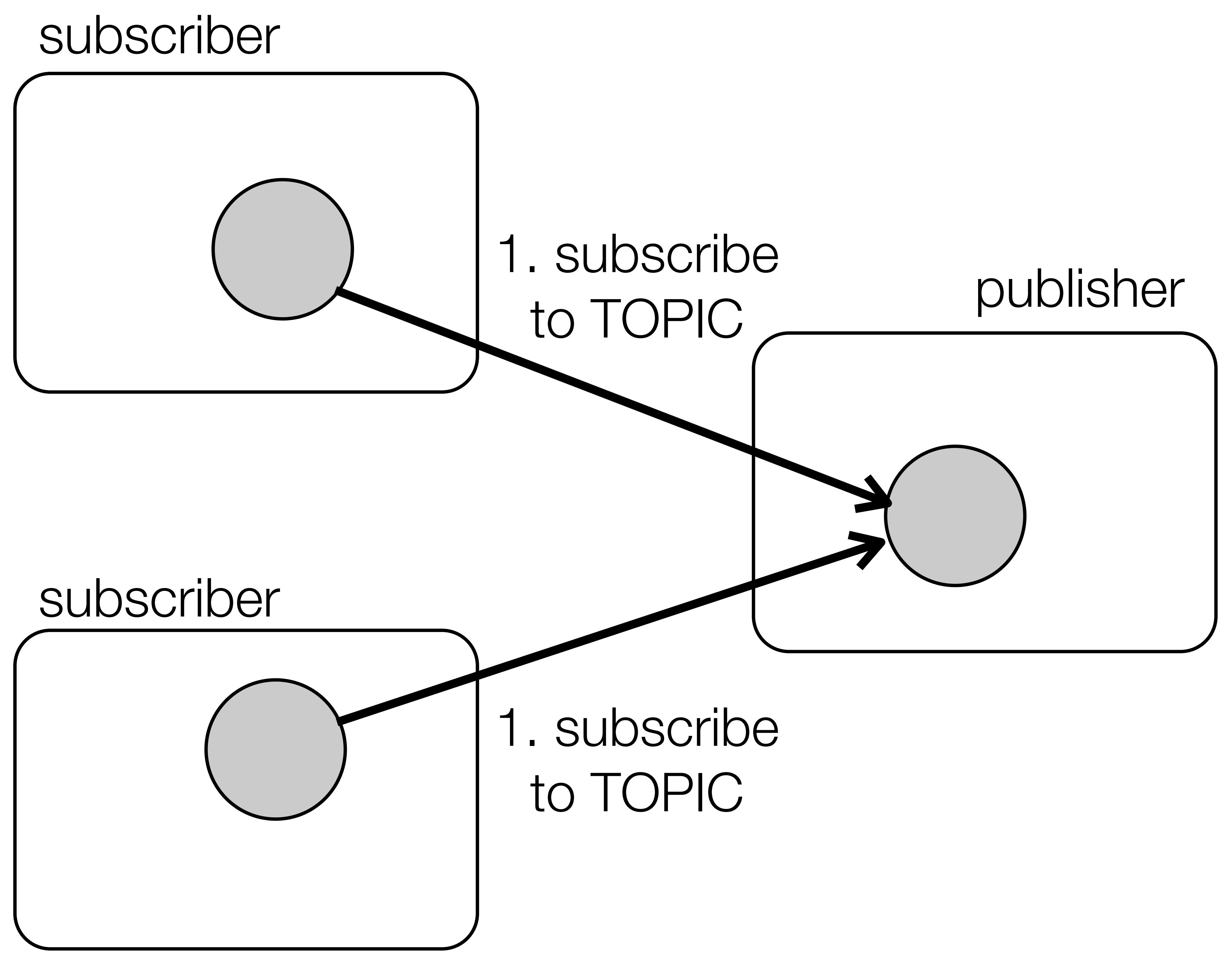
$\Bigg\}$ Direct communication (time and space-coupled)

- Publish-subscribe
- Message queue
- Tuple space

$\Bigg\}$ Indirect communication (time and space de-coupled)

Event notification

- Subscribers connect directly to a single “source” publisher, optionally indicating a “topic” of interest
- The publisher broadcasts events to all connected subscribers that subscribed to the event topic

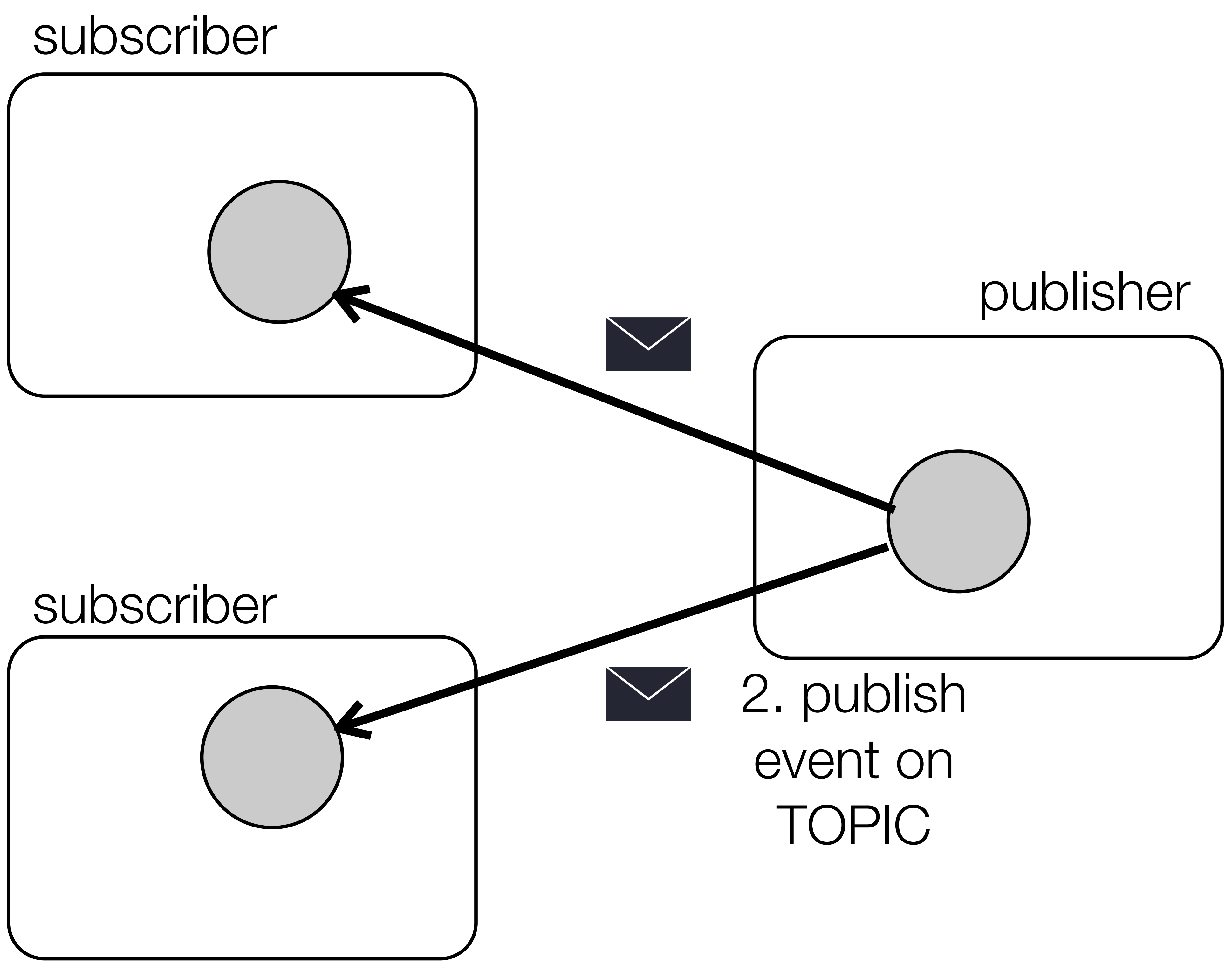


ØMQ

/event_notification

Event notification

- Subscribers connect directly to a single “source” publisher, optionally indicating a “topic” of interest
- The publisher broadcasts events to all connected subscribers that subscribed to the event topic



ØMQ

/event_notification

Event notification: summary

- Time-coupled: subscriber must be online at the time an event is published
- Space-coupled: subscribers must know the publisher
- Supports one-to-many communication
- Subscribers listen for events, publisher usually doesn't wait for a reply from the subscibers ("fire-and-forget")
- The topic name acts as a filter to only notify subscribers of relevant events
- Useful pattern for real-time information dissemination (think: sensor readings in an IoT app, ticker tape updates in a financial trading app)

Communication Patterns

- Request-reply
- Event Notification
- **Pipeline (push-pull)**

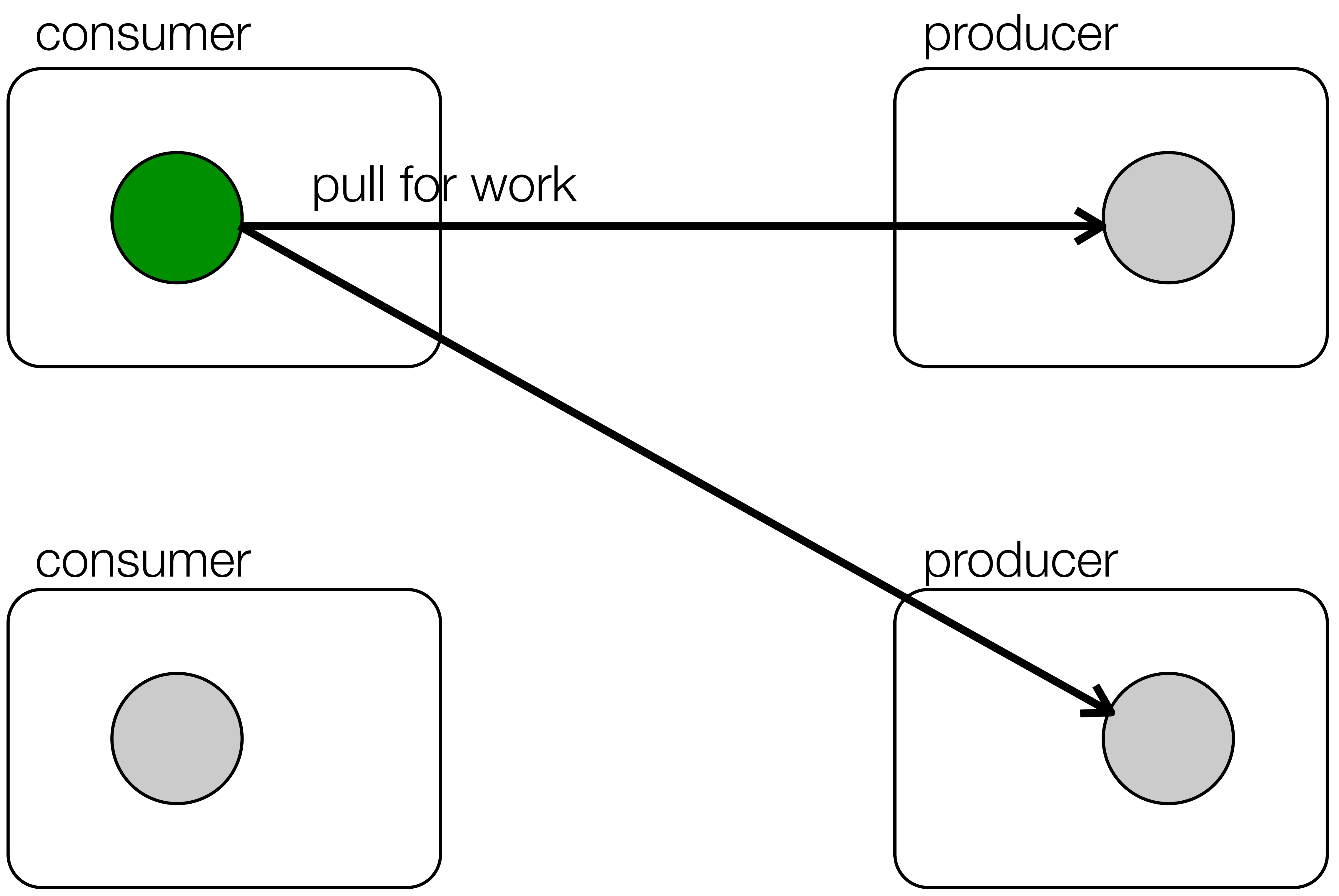
$\Big\}$ Direct communication (time and space-coupled)

- Publish-subscribe
- Message queue
- Tuple space

$\Big\}$ Indirect communication (time and space de-coupled)

Pipeline (push-pull)

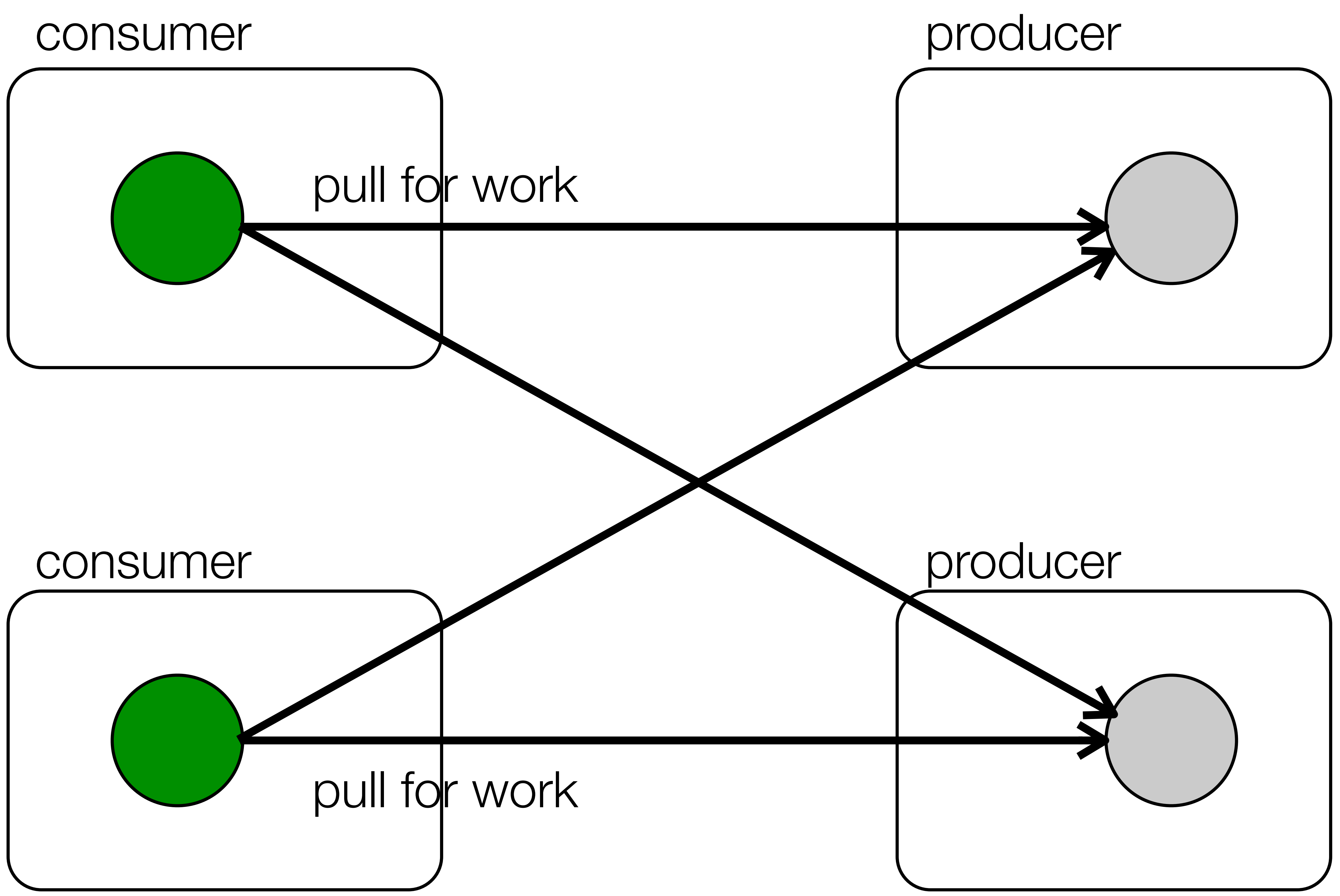
- Consumers pull work from one or more producers, waiting until one of them is available to push
- Producers push work to one or more consumers, waiting until one of them is available to pull



ØMQ /pipeline

Pipeline (push-pull)

- Consumers pull work from one or more producers, waiting until one of them is available to push
- Producers push work to one or more consumers, waiting until one of them is available to pull

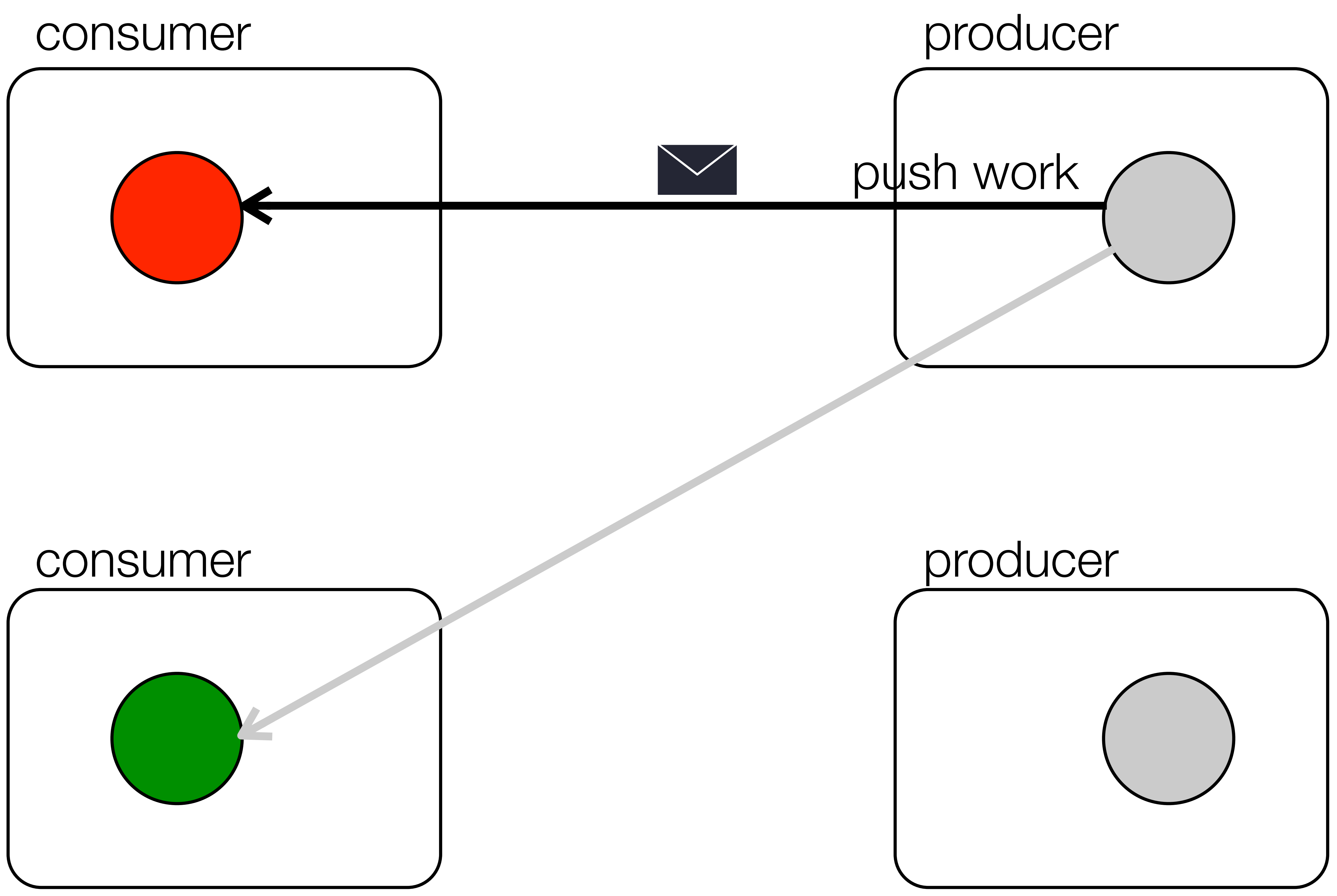


ØMQ

/pipeline

Pipeline (push-pull)

- Consumers pull work from one or more producers, waiting until one of them is available to push
- Producers push work to one or more consumers, waiting until one of them is available to pull

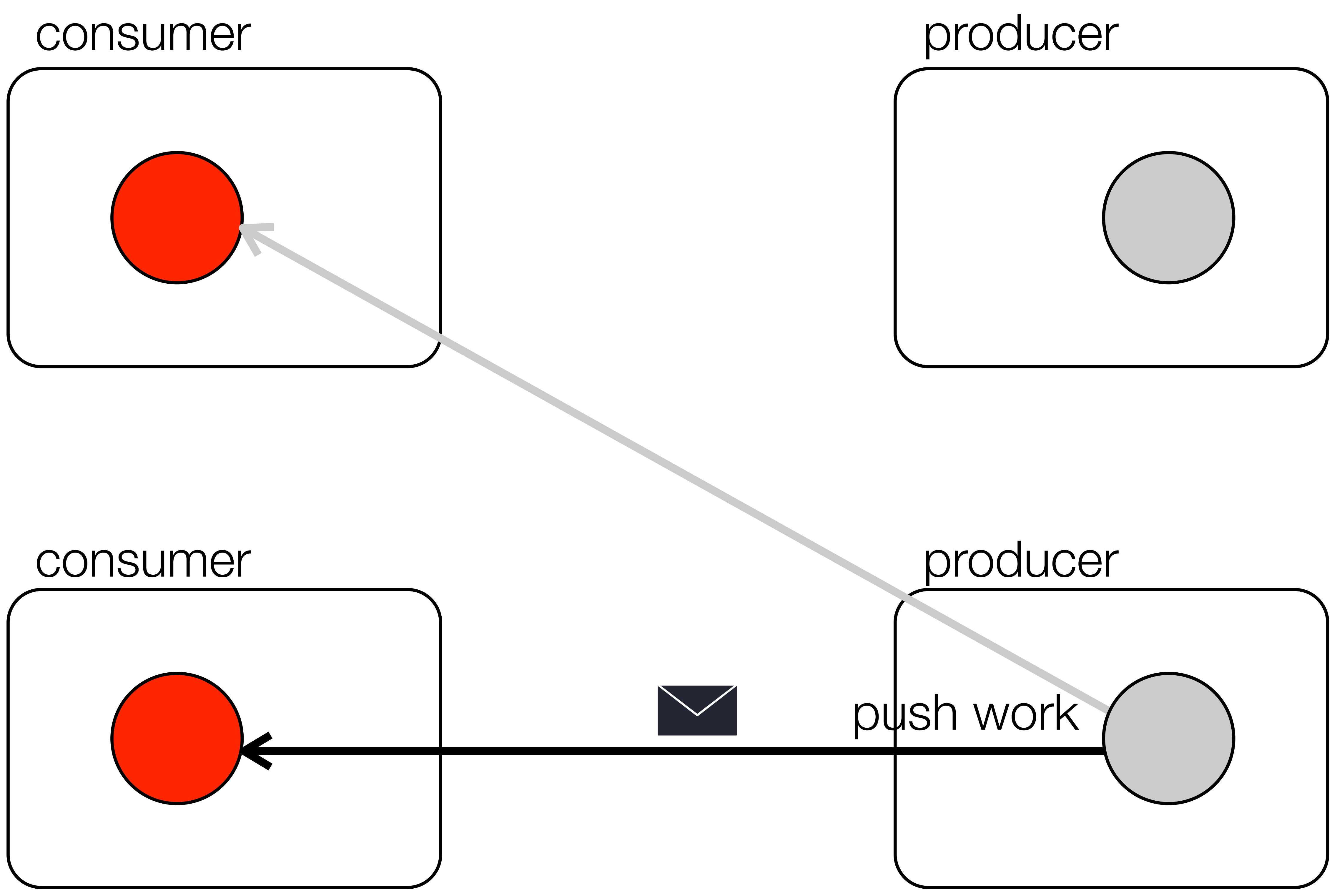


ØMQ

/pipeline

Pipeline (push-pull)

- Consumers pull work from one or more producers, waiting until one of them is available to push
- Producers push work to one or more consumers, waiting until one of them is available to pull



ØMQ
/pipeline

KU LEUVEN DistriNet

Pipeline (push-pull): summary

- Time-coupled: consumers and producers must both be online to push or pull work
- Space-coupled: consumers connect directly to producers, producers listen
- Supports many-to-many communication
- **Synchronous**: consumers block if no producer pushes work, and producers block if no consumer pulls for work
- Communication is usually **one-way**. There is no natural way for the consumer to communicate the result of the task back to the producer.
- Useful pattern to **load-balance work** (tasks) across many “workers”: each pushed message gets routed to a single consumer who is responsible to handle it.
- If multiple consumers are available to pull work, a **fair queueing strategy** ensures that work is divided evenly across consumers (e.g. hand out the work in a round-robin fashion)

Communication Patterns

- Request-reply
- Event Notification
- Pipeline (push-pull)

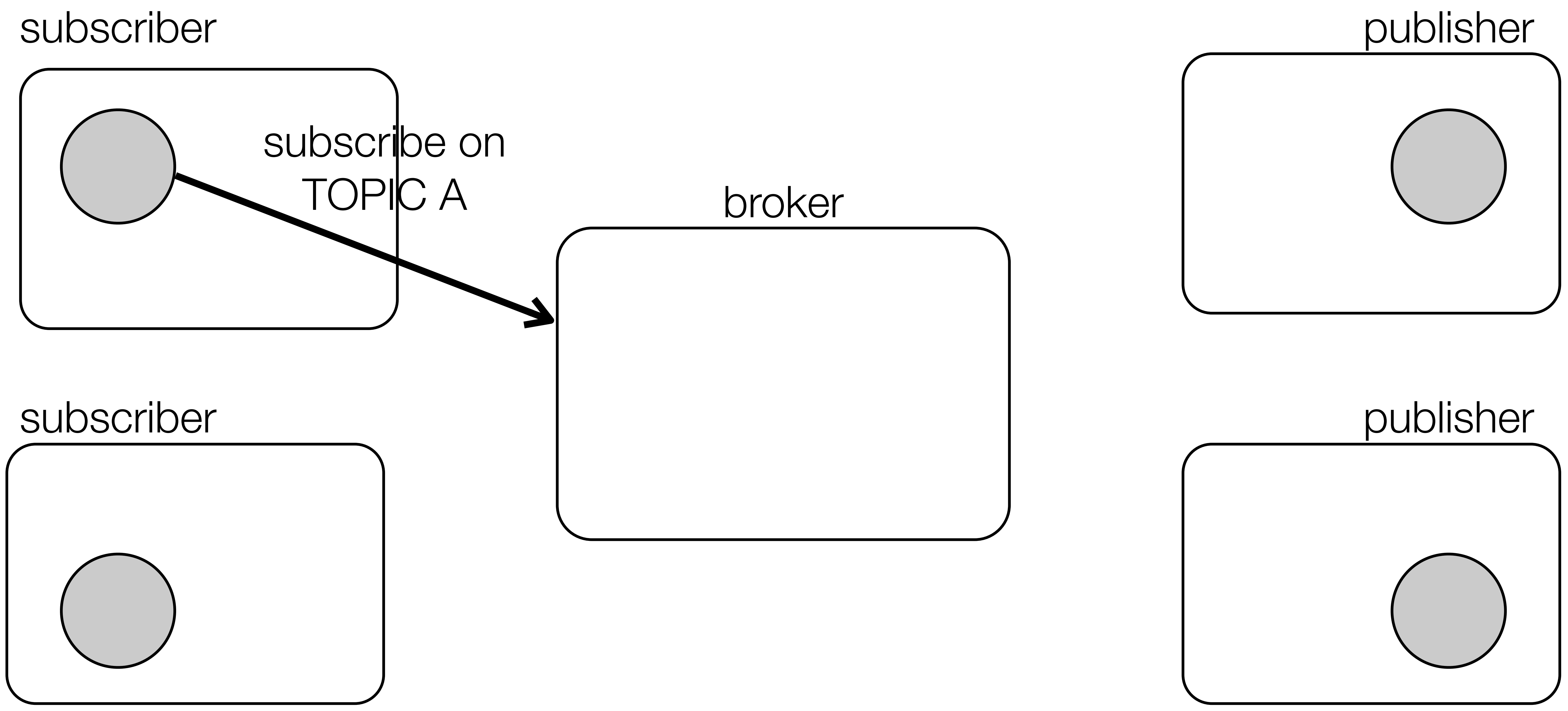
} Direct communication (time and space-coupled)

- **Publish-subscribe**
- Message queue
- Tuple space

} Indirect communication (time and space de-coupled)

Publish-subscribe

- Subscribers connect to a shared “message broker” or “event bus”, indicating a topic of interest. Optionally, they may create a queue that will store any event notifications while the subscriber is unavailable.
- Publishers also connect to the broker. Publishers send event notifications to the broker, who forwards them to the subscribers (or queues the notification if the subscriber is unavailable).

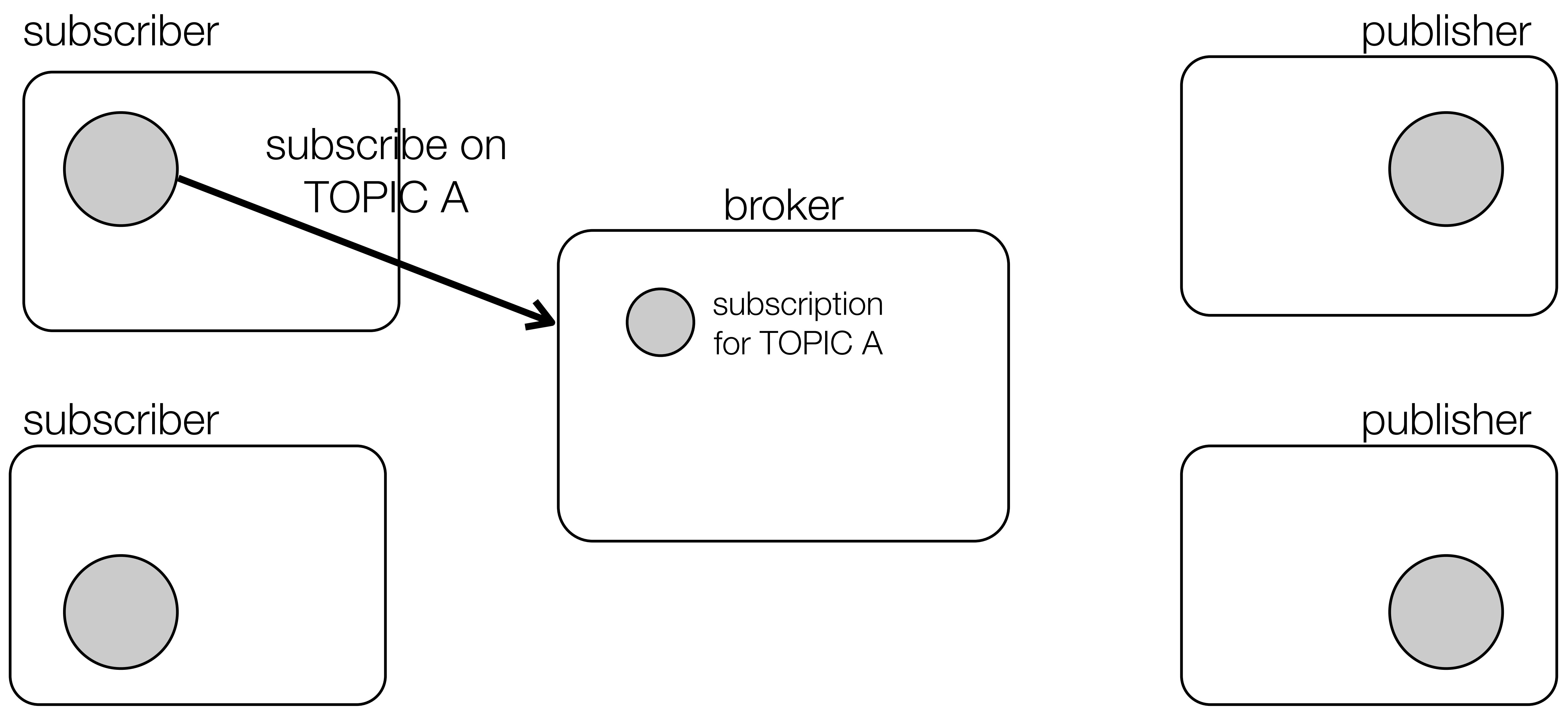


RabbitMQ

/publish_subscribe

Publish-subscribe

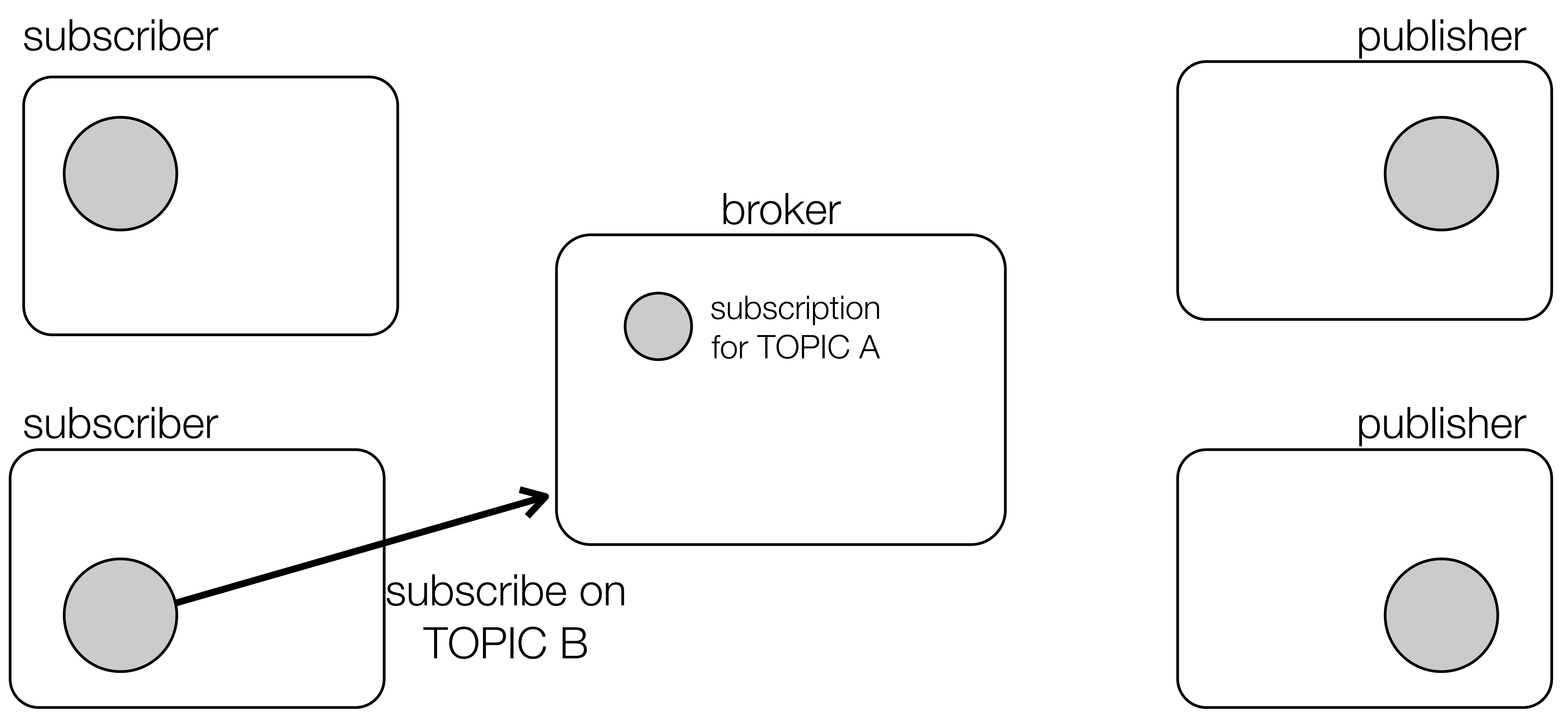
- Subscribers connect to a shared “message broker” or “event bus”, indicating a topic of interest.
- Optionally, they may create a queue on the broker that will store any event notifications while the subscriber is unavailable.



RabbitMQ

/publish_subscribe

Publish-subscribe

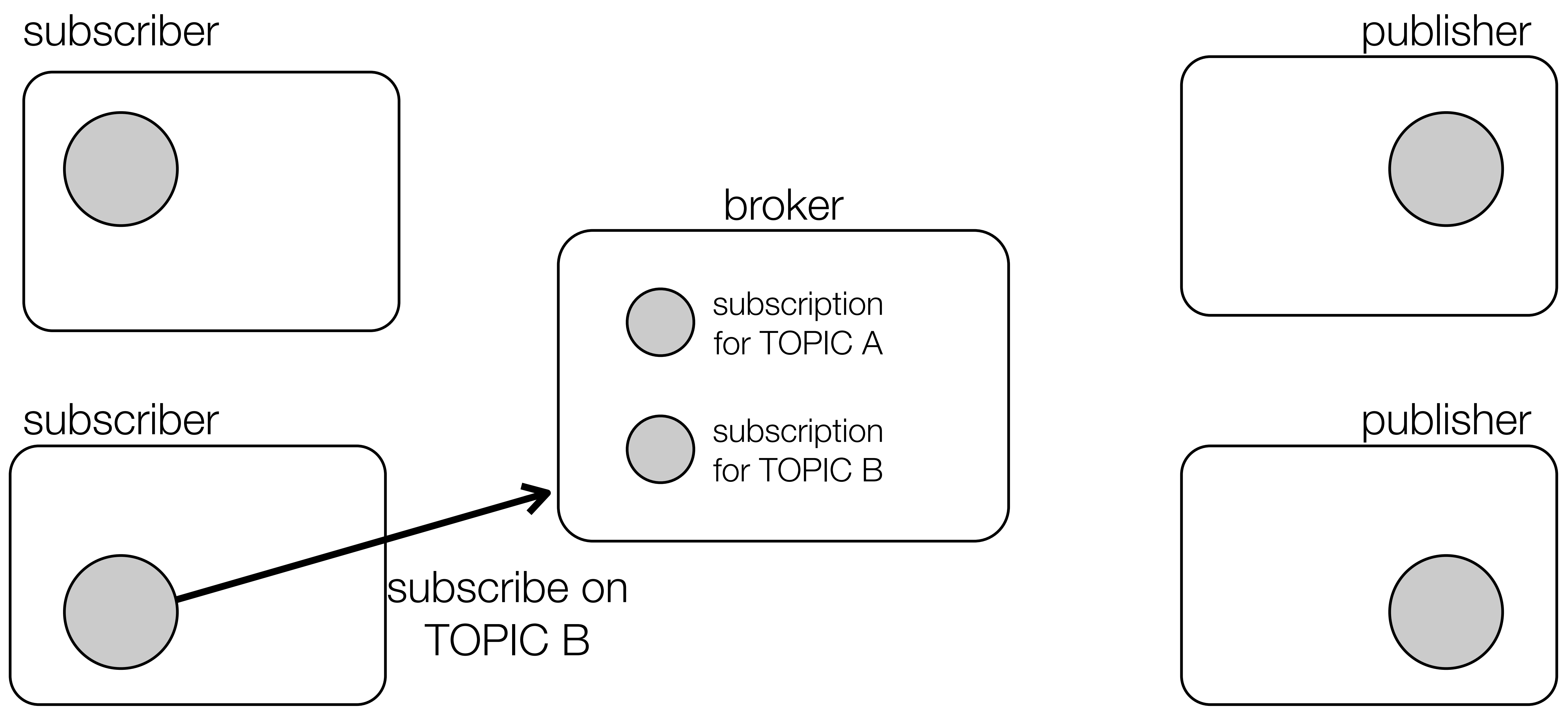


RabbitMQ

/publish_subscribe

KU LEUVEN DistriNet

Publish-subscribe

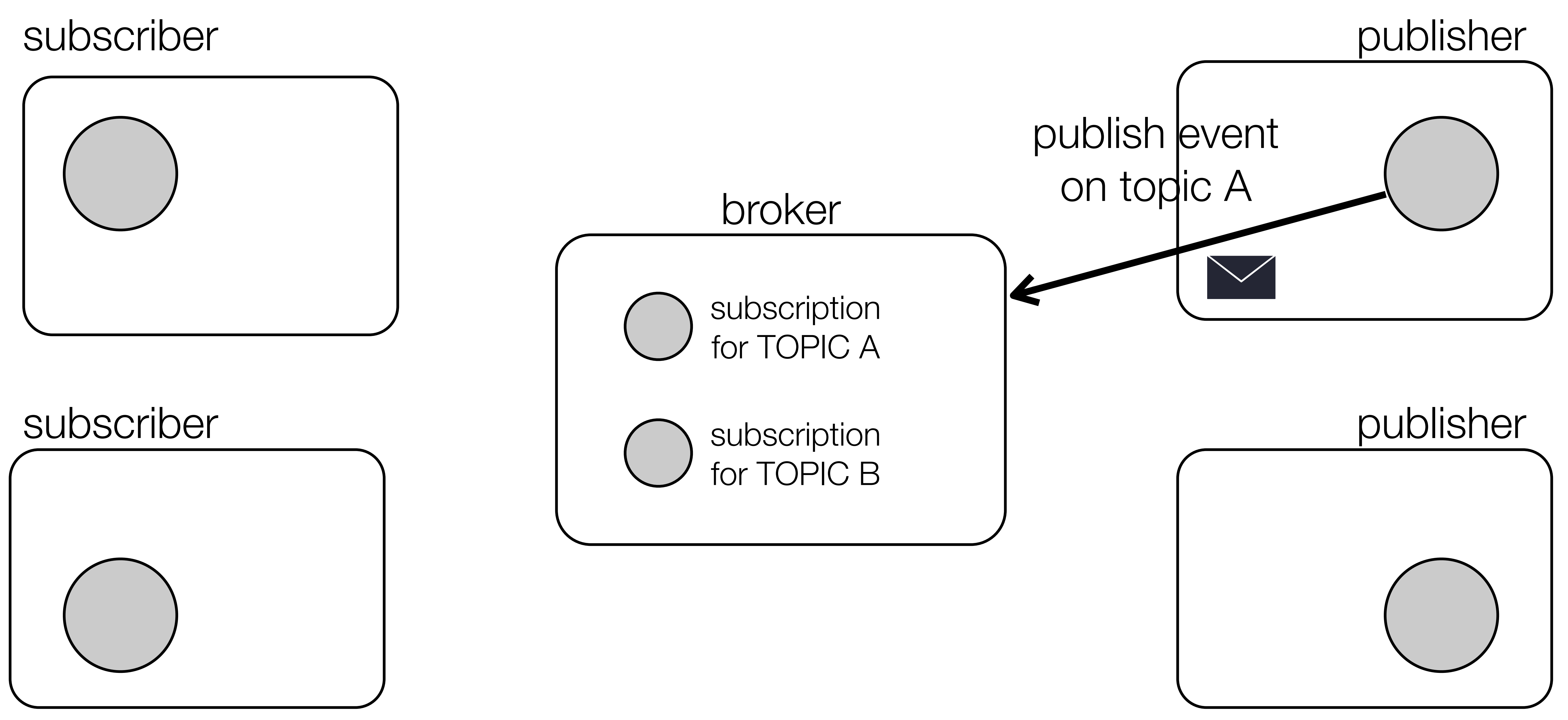


RabbitMQ

/publish_subscribe

Publish-subscribe

- Publishers also connect to the broker. Publishers send event notifications to the broker, directed to a specific topic.
- The broker forwards the event to the right subscribers (or queues the notification if the subscriber is unavailable and created a queue)



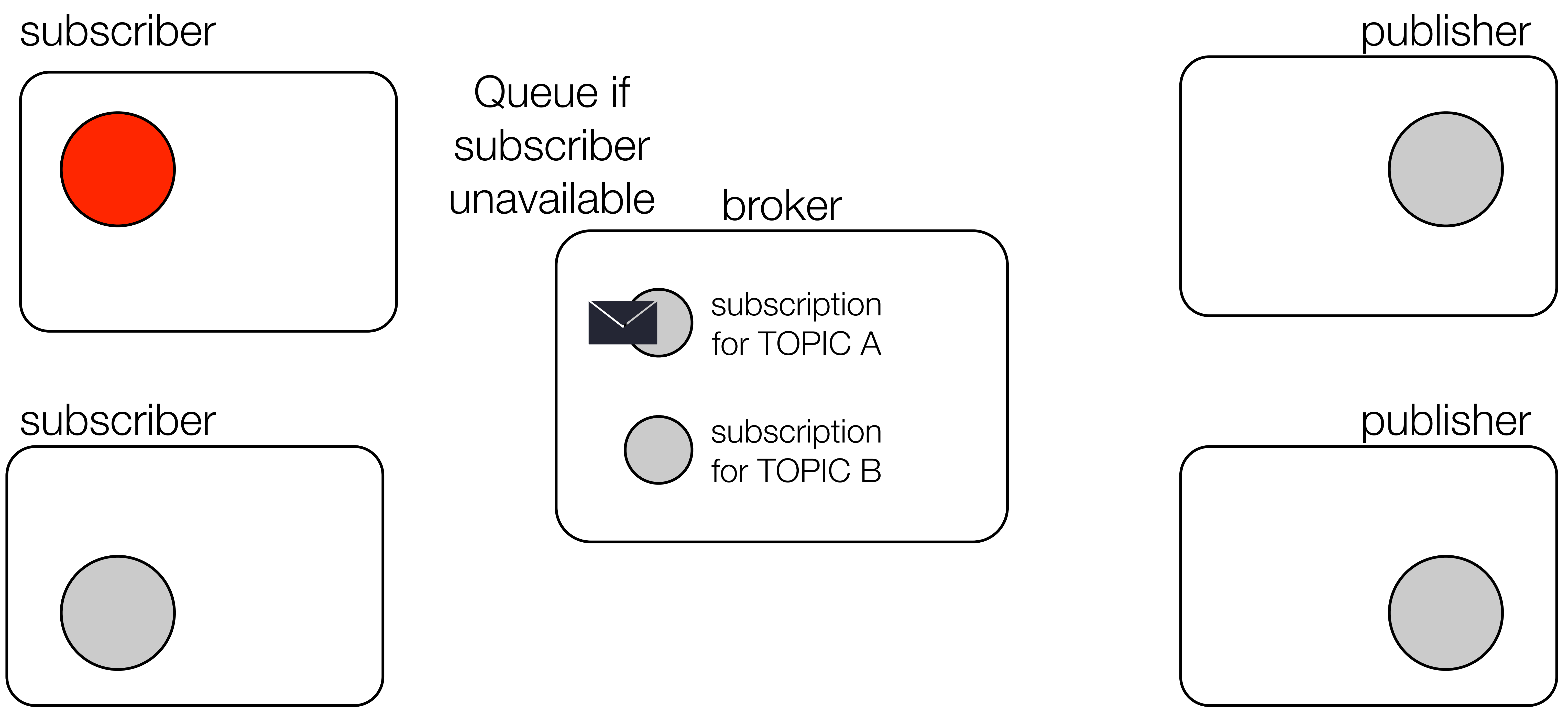
RabbitMQ

/publish_subscribe

KU LEUVEN DistriNet

Publish-subscribe

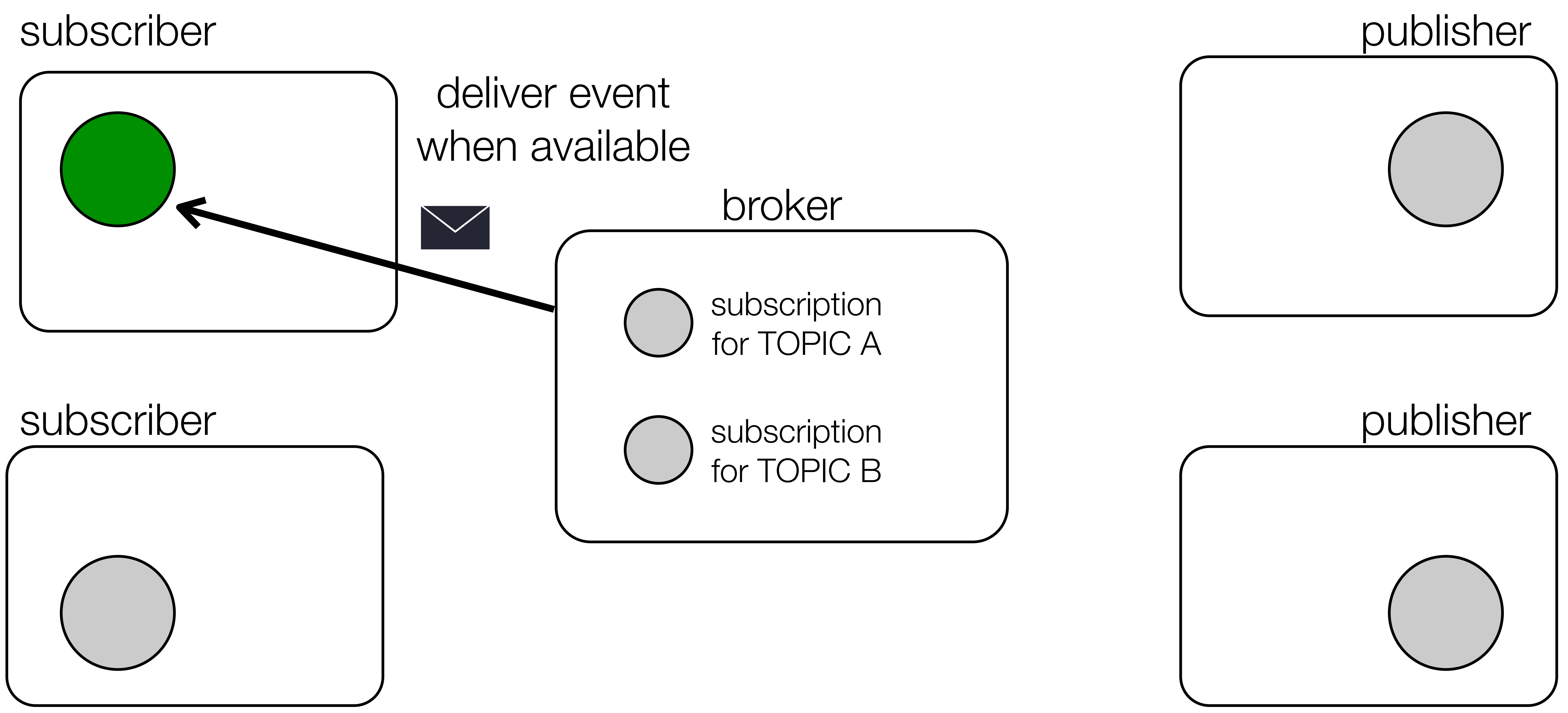
- Publishers also connect to the broker. Publishers send event notifications to the broker, directed to a specific topic.
- The broker forwards the event to the right subscribers (or queues the notification if the subscriber is unavailable and created a queue)



/publish_subscribe

Publish-subscribe

- Publishers also connect to the broker. Publishers send event notifications to the broker, directed to a specific topic.
- The broker forwards the event to the right subscribers (or queues the notification if the subscriber is unavailable and created a queue)



/publish_subscribe

Publish-subscribe

- How are event subscriptions matched with event notifications?
- The simplest approach is to use a shared topic name. This is called **topic-based** publish-subscribe.
  - Sometimes the topic name can be hierarchical. For example, AMQP supports topic names like `news.sports.football` or `news.sports.tennis` and subscriptions with wildcards that match any subtopic. For example, to receive all sports news, a subscriber can subscribe to the topic `news.sports.*`
- Another approach is to match on the content of the event itself. This is called **content-based** publish-subscribe.
  - Events modeled as attribute-value pairs (e.g. `{“city”=“Leuven”, “weather”=“sunny”, “temp”=32, “date”=“01-08-2023”}`)
  - Subscriptions can be complex predicates on attribute *values* (e.g. `“city”=* AND “temp” > 28 AND “date” OLDER THAN “01-01-2023”`)
  - More flexible, but also more complex to identify matching subscriptions.

Publish-subscribe: summary

- Time-decoupled: the broker can store messages on behalf of subscribers. If a subscriber does not create a queue, then it must remain connected to the broker to receive event notifications (not time-decoupled).
- Space-decoupled: publishers and subscribers do not need to know each other's identity, they all connect to a shared broker instead.
- Supports many-to-many communication
- Publishers do not wait for a response from subscribers. If a response is needed, subscribers must send an explicit reply (e.g. on a topic that the publisher is listening on).
- Subscribers can either *poll* their queue periodically or open a channel on which to receive *push notifications* from the broker when new events arrive (typically triggering a *callback* or *listener* in the subscriber code).
- To guarantee message delivery, subscribers must *ack* receipt of the event notification to the broker so that the broker can delete it from the queue.
- Useful pattern to facilitate **enterprise application integration** (EAI): allows many enterprise services to interact in a loosely coupled way, making it easy to add, remove or replace services without affecting all the other services.

Communication Patterns

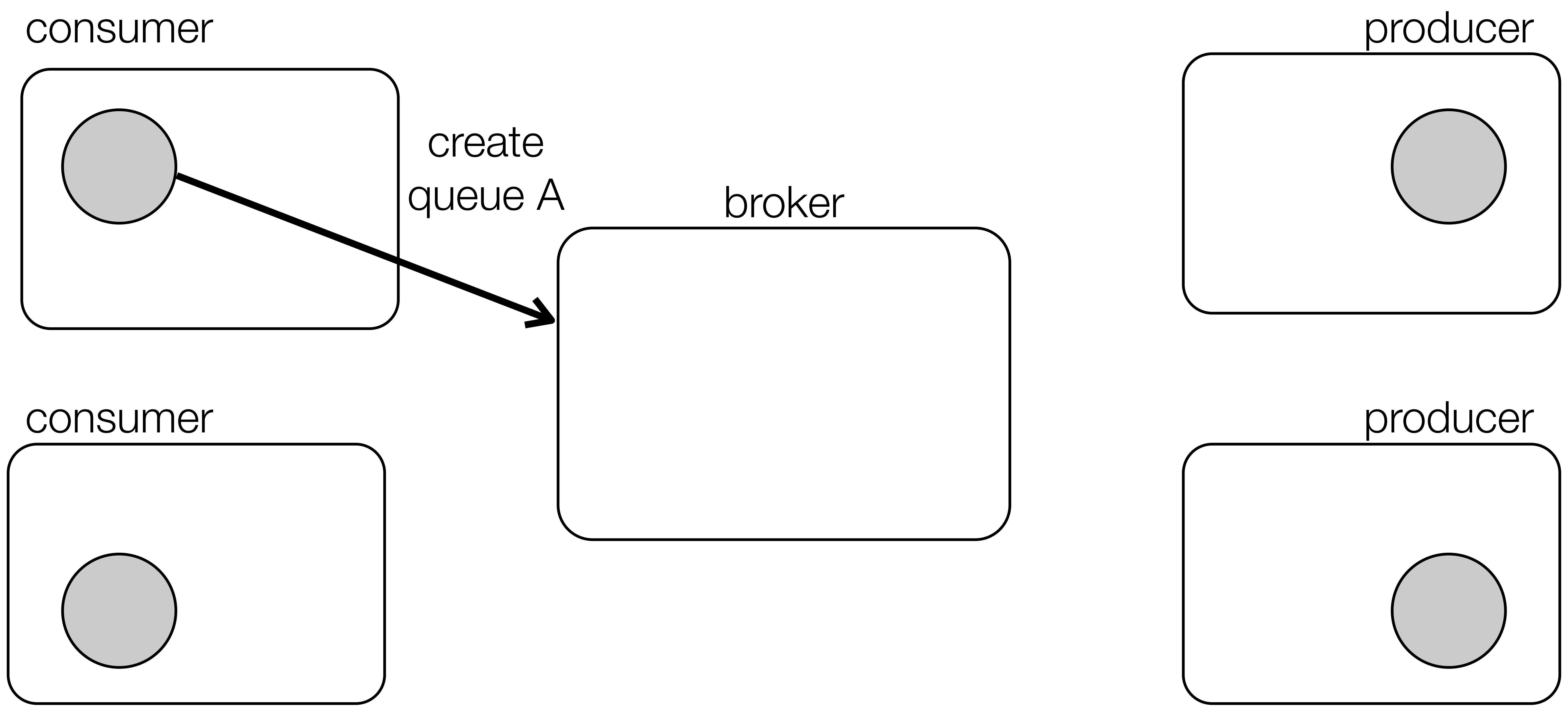
- Request-reply
- Event Notification
- Pipeline (push-pull)
- Publish-subscribe
- **Message queue**
- Tuple space

$\left.\begin{array}{l}\text{Request-reply}\\\text{Event Notification}\\\text{Pipeline (push-pull)}\end{array}\right\}$ Direct communication (time and space-coupled)

$\left.\begin{array}{l}\text{Publish-subscribe}\\\text{Message queue}\\\text{Tuple space}\end{array}\right\}$ Indirect communication (time and space de-coupled)

Message Queueing

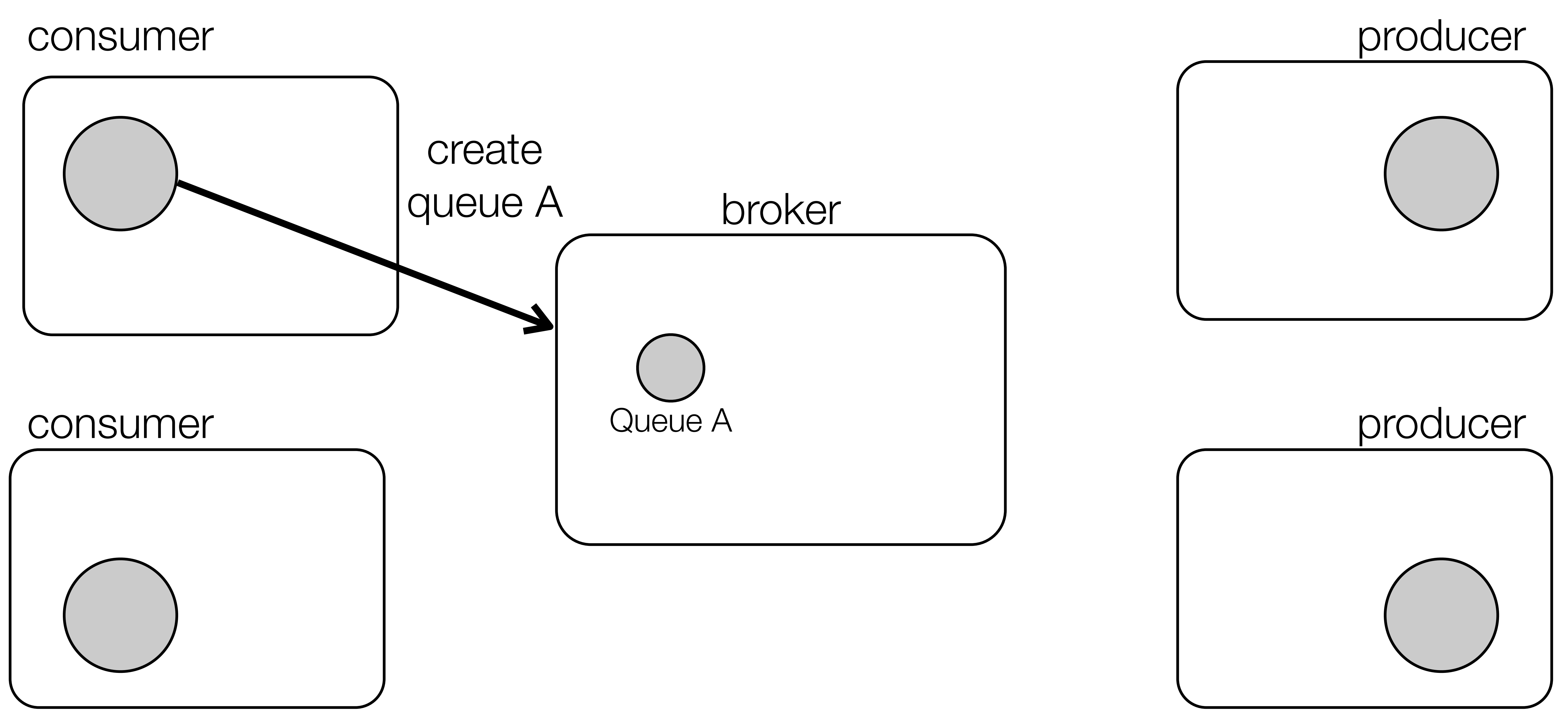
- Similar to the Pipeline pattern, but with the broker acting as an intermediary.
- Key difference: tasks can be queued in the broker. Tasks are not deleted from the queue until they are explicitly acknowledged by the consumer. This makes task execution resilient to failures!



/message_queue

Message Queueing

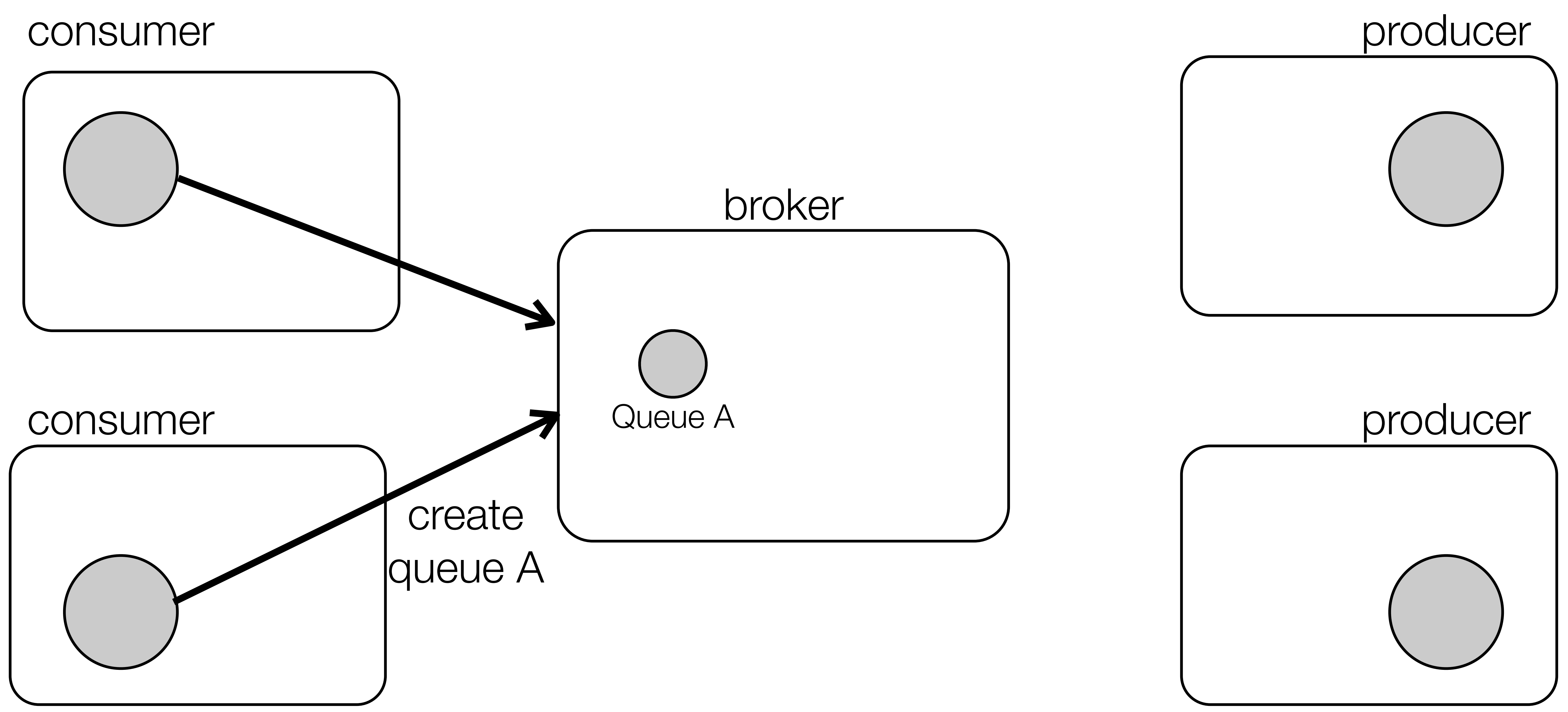
- Consumers can create FIFO queues on the broker, and then wait for messages in the queue



/message_queue

Message Queueing

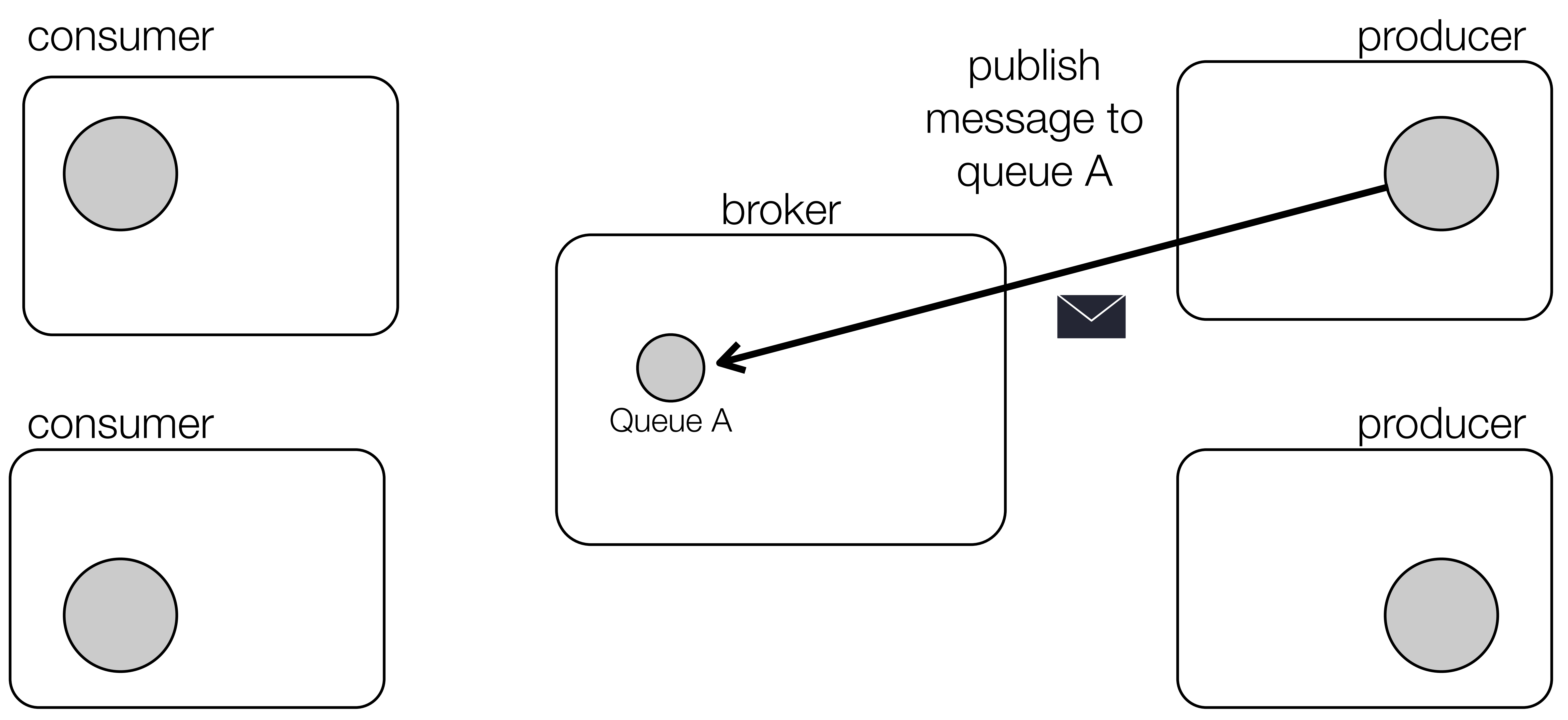
- Multiple consumers can register to the same queue!



/message_queue

Message Queueing

- Producers can send ("publish") messages to a named queue



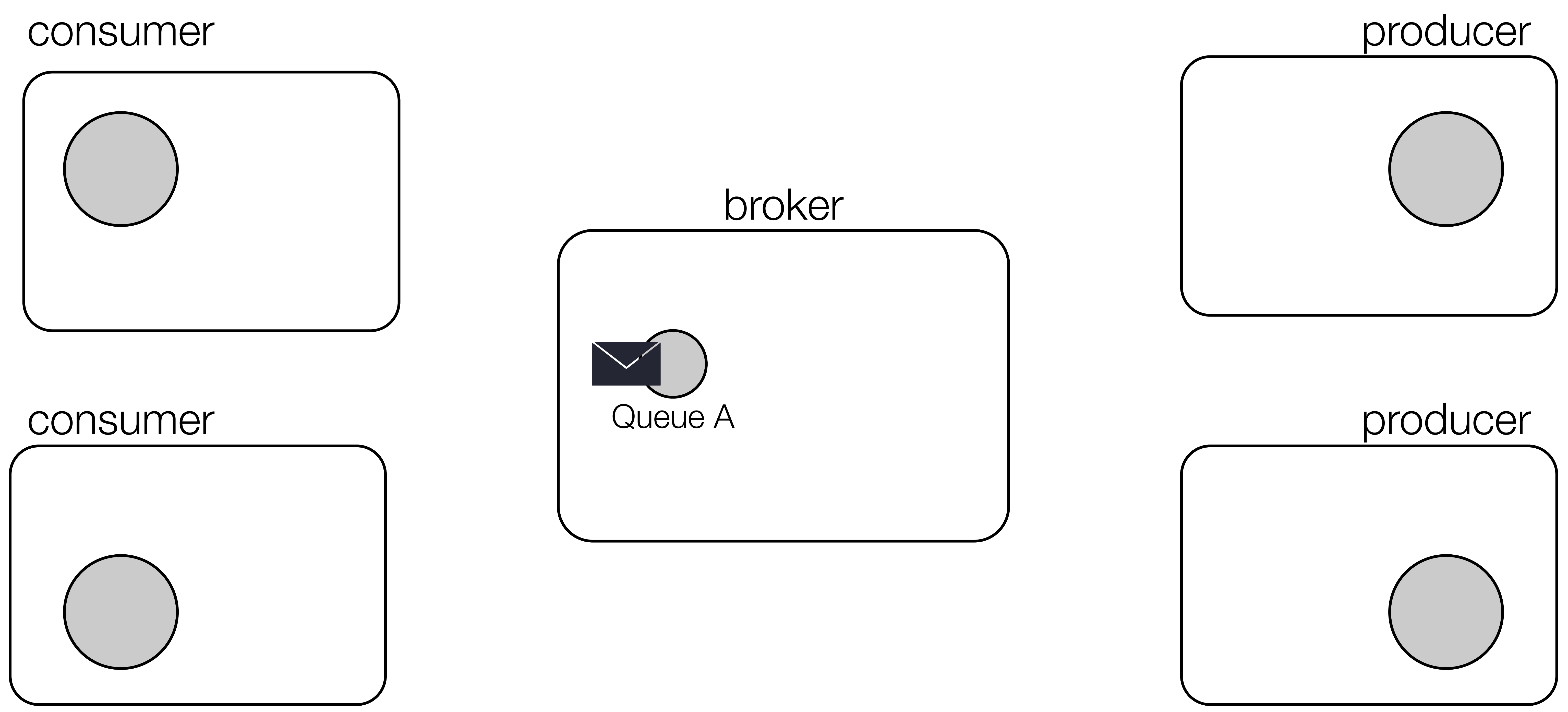
RabbitMQ

/message_queue

KU LEUVEN DistriNet

Message Queueing

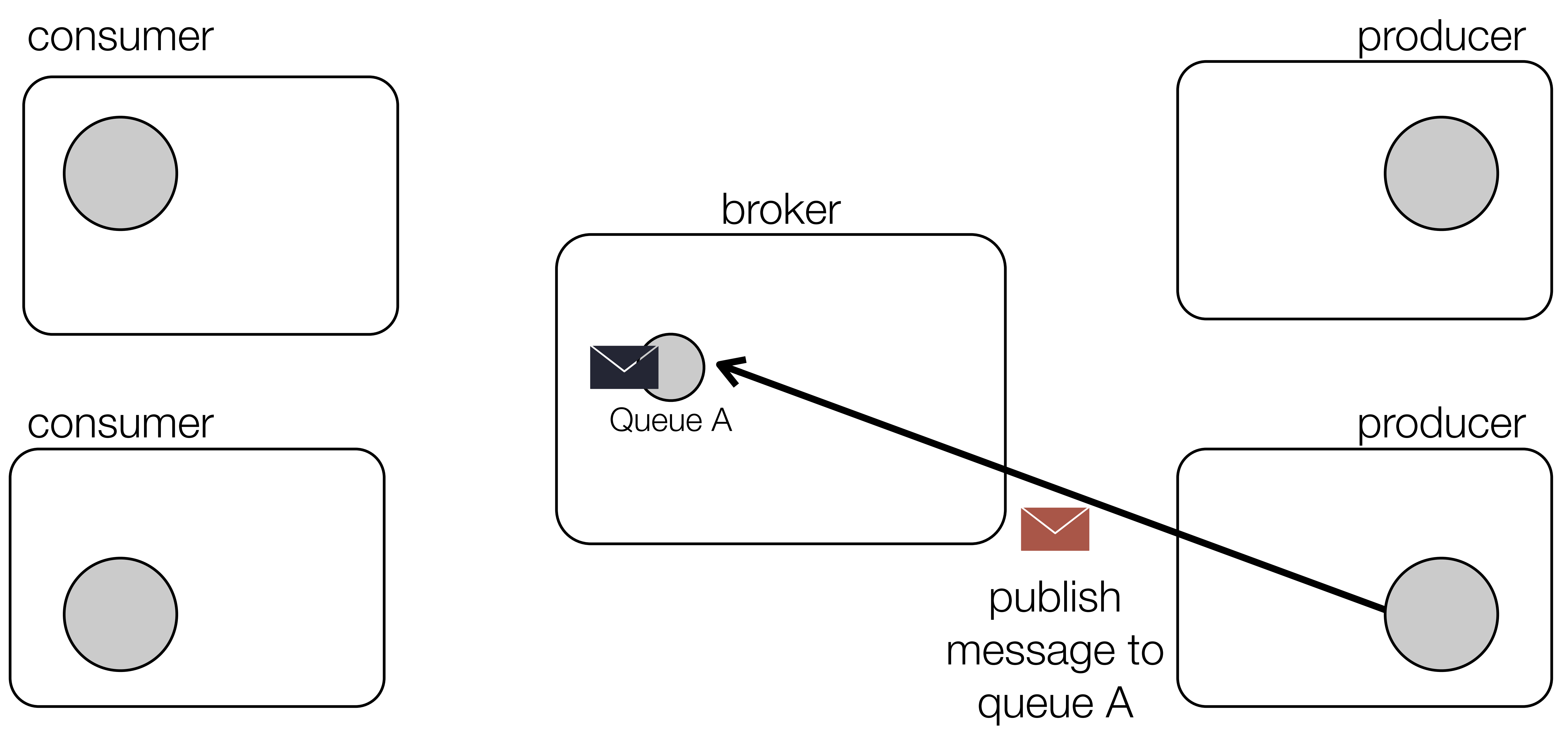
- Messages are enqueued in order of arrival at the broker and can be consumed in FIFO order



/message_queue

Message Queueing

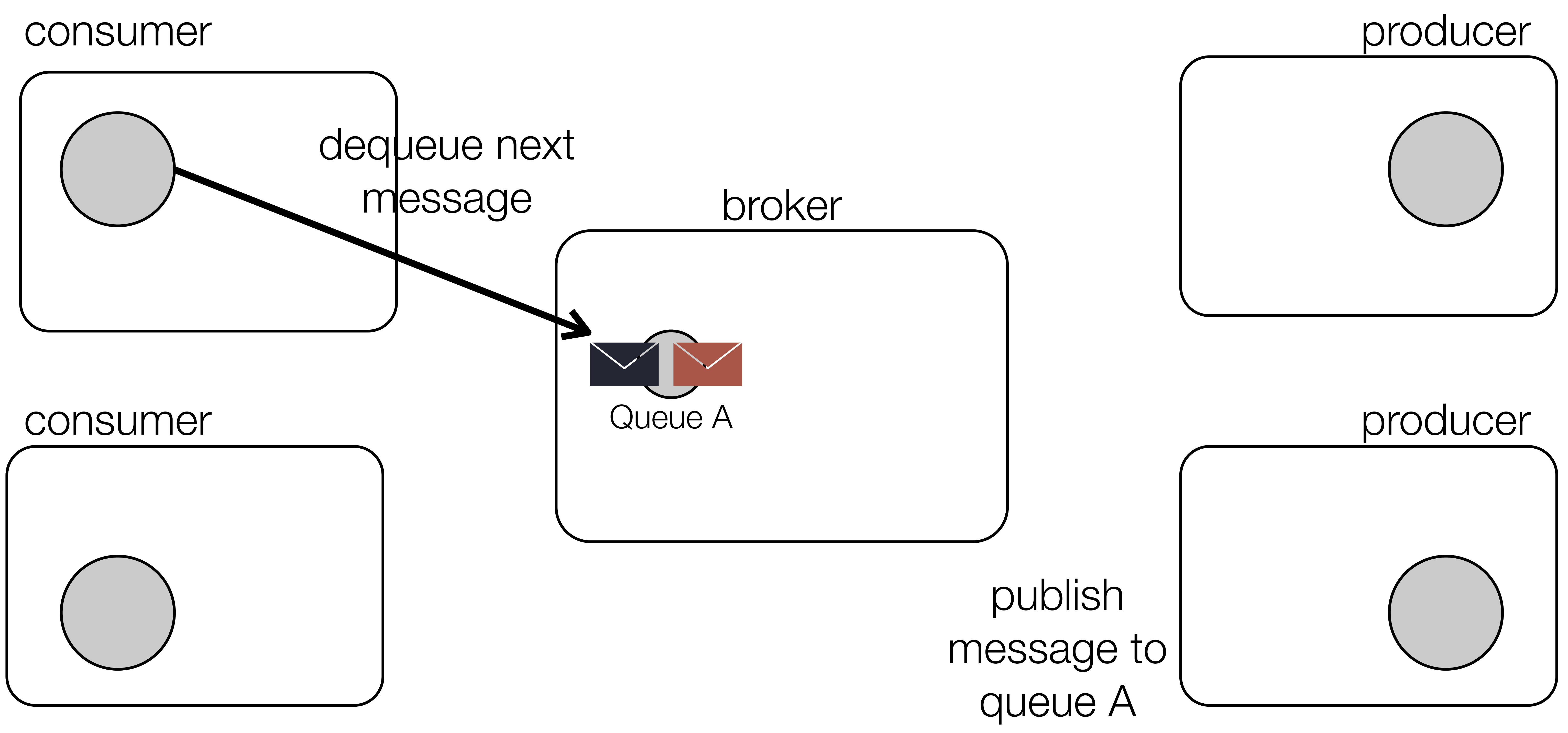
- Multiple producers can send messages to the same queue



/message_queue

Message Queueing

- Consumers can poll the queue for the next message.



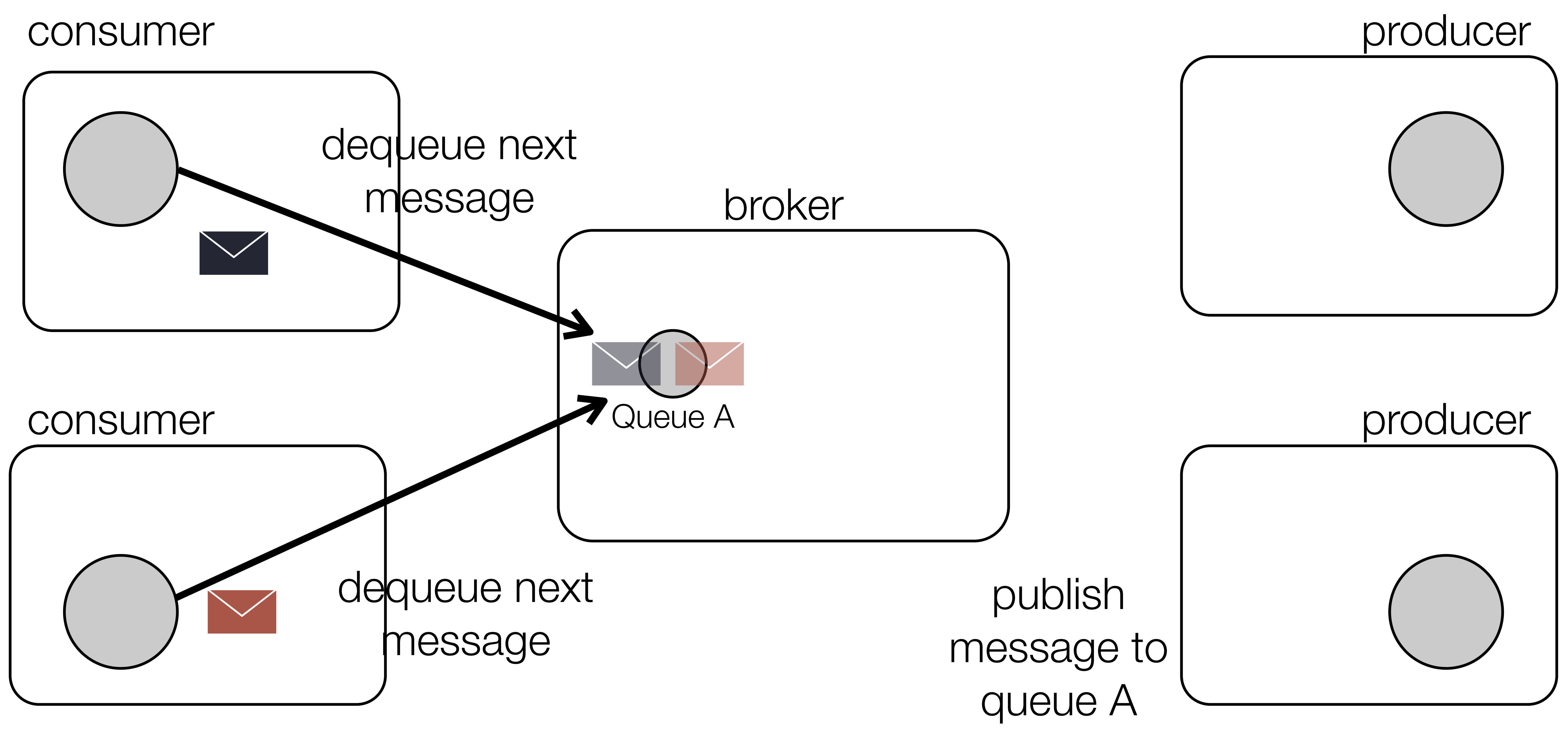
RabbitMQ

/message_queue

KU LEUVEN DistriNet

Message Queueing

- If multiple consumers try to dequeue simultaneously, the broker ensures each message is delivered to at most one consumer



/message_queue

Message Queueing: summary

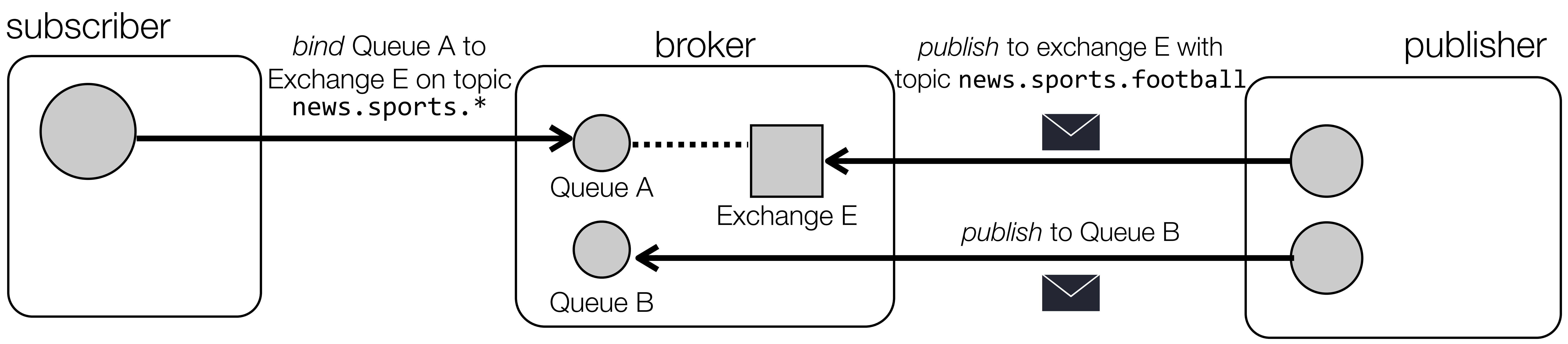
- Time-decoupled: the broker can store messages on behalf of consumers when they are unavailable.
- Space-decoupled: consumers and producers do not need to know each other's identity, they all connect to a shared broker (and queue) instead.
- Supports many-to-many communication
- **Message Queueing versus Publish-Subscribe**
  - In Publish-Subscribe each subscriber is typically independent and each message (event) is sent to *all* matching subscribers (*dedicated* queue per subscriber)
  - In Message Queueing the consumers are "competing" for the same tasks and each message (event) is delivered to only *one* consumer (*shared* queue for all consumers).
- **Message Queueing versus Pipeline (Push-Pull)**
  - Apart from the fact that the Pipeline pattern is not decoupled in time and space, another important difference is that in message queueing, the producers do not wait for consumers to be available, but instead just deliver the message to the broker.
- Useful pattern to allocate tasks to different workers, especially if it is important that tasks are not lost if a worker crashes.

Message-oriented Middleware (MOM)

- Middleware that provides distributed message queues.
- Supports both Publish-Subscribe and Message Queuing communication styles.
- Often supports advanced features such as reliable message delivery, persistent queues, message priorities, broker federation, etc.

Side-note: AMQP

- Advanced Message Queuing Protocol (**AMQP**) is a standardised protocol for message brokers.
- Multiple message brokers that implement AMQP exist (e.g. RabbitMQ, Apache ActiveMQ, …)
- AMQP allows for the creation of **exchanges** and **queues** on the broker.
- A queue **binds** to an exchange with a given **topic**.
- Events can be sent to an exchange (for delivery to all queues with a matching topic) or to a queue directly.


subscriber
bind Queue A to Exchange E on topic news.sports.*
broker
Queue A
Exchange E
Queue B
publish to exchange E with topic news.sports.football
publish to Queue B
publisher

Communication Patterns

- Request-reply
- Event Notification
- Pipeline (push-pull)

$\}$ Direct communication (time and space-coupled)

- Publish-subscribe
- Message queue
- **Tuple space**

$\}$ Indirect communication (time and space de-coupled)

Tuple Spaces

- Introduced in the Linda coordination language (Gelernter & Carriero, 1980s).
- Influential on distributed systems architects, but not widely used.
- Processes communicate via a **logically shared** memory space: the tuple space
- Processes insert and remove **tuples** from the tuple space
- Adding tuples is asynchronous
- Removing tuples is usually synchronous

Coordination Languages and their Significance

We can build a complete programming model out of two separate pieces—the *computation model* and the *coordination model*. The computation model allows programmers to build a single computational activity: a single-threaded, step-at-a-time computation. The coordination model is the glue that binds separate activities into an ensemble. An ordinary computation language (e.g., Fortran) embodies some computation model. A coordination language embodies a coordination model; it provides operations to *create* computational activities and to support *communication* among them.

Our approach to coordination has been developed in the framework of a system called Linda.™ Linda is not a programming language. Kahn and Miller write that "Linda is best not thought of as a language—but rather as an extension that can be added to nearly any language to enable process creation, communication, and synchronization[27]." We would rather say that Linda is a coordination language. It is one of two components that together make up a *complete* programming language. (The suggestion that traditional programming languages are *incomplete* is intentional.)

A computation model and a coordination model might be integrated into a single language. They might also be separated into two distinct languages, in which case programmers choose one of each: one computation language plus

David Gelernter
Nicholas Carriero

COMMUNICATIONS OF THE ACM/February 1992/Vol.35, No.2

What is a Tuple?

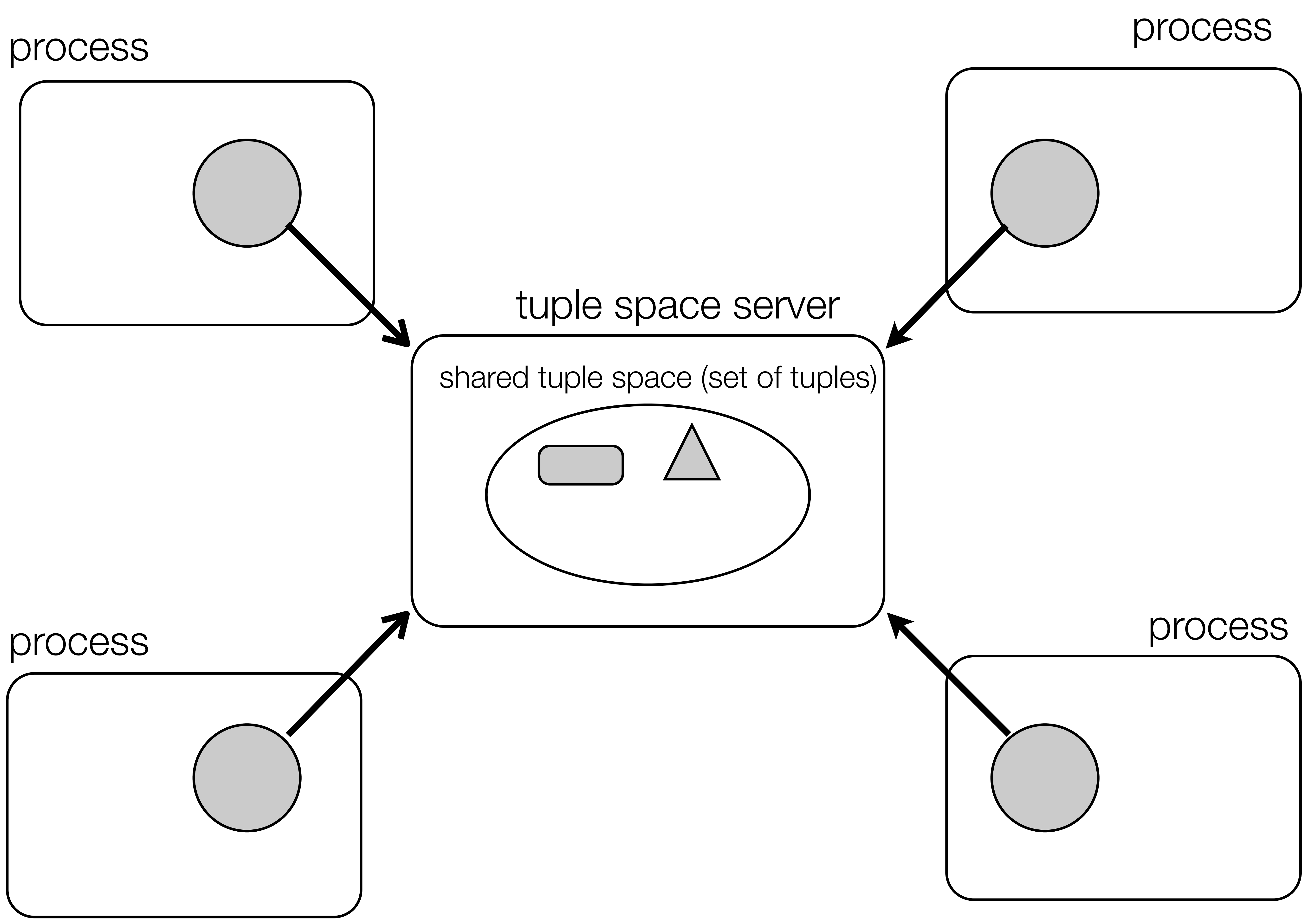
- An ordered sequence of typed data values
- Producer inserts tuples with concrete values

```
{“john”, 42}
```

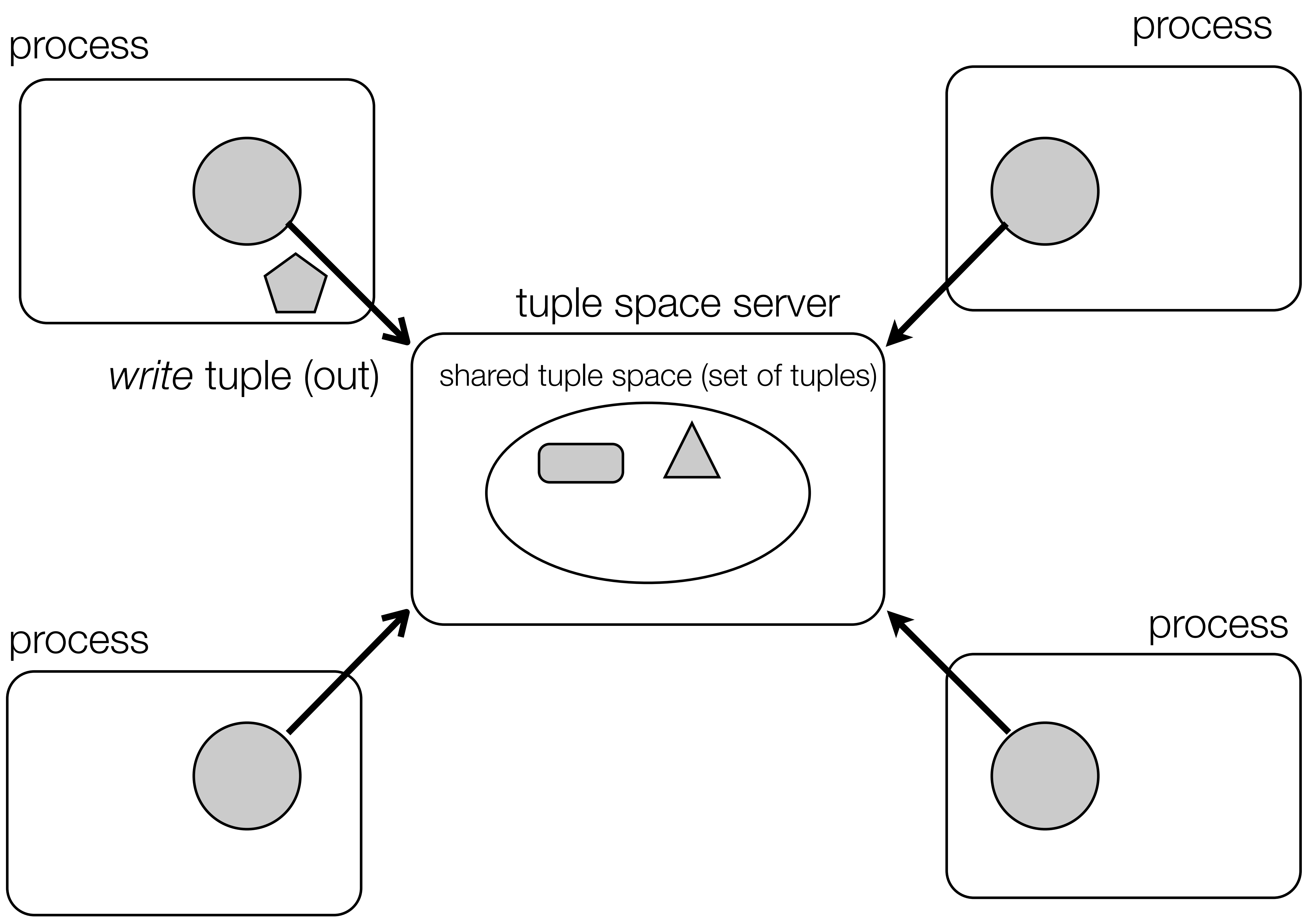
- Consumer reads tuples based on a *template* (which may contain *wildcards*)

```
{ “john”, ?age }
```

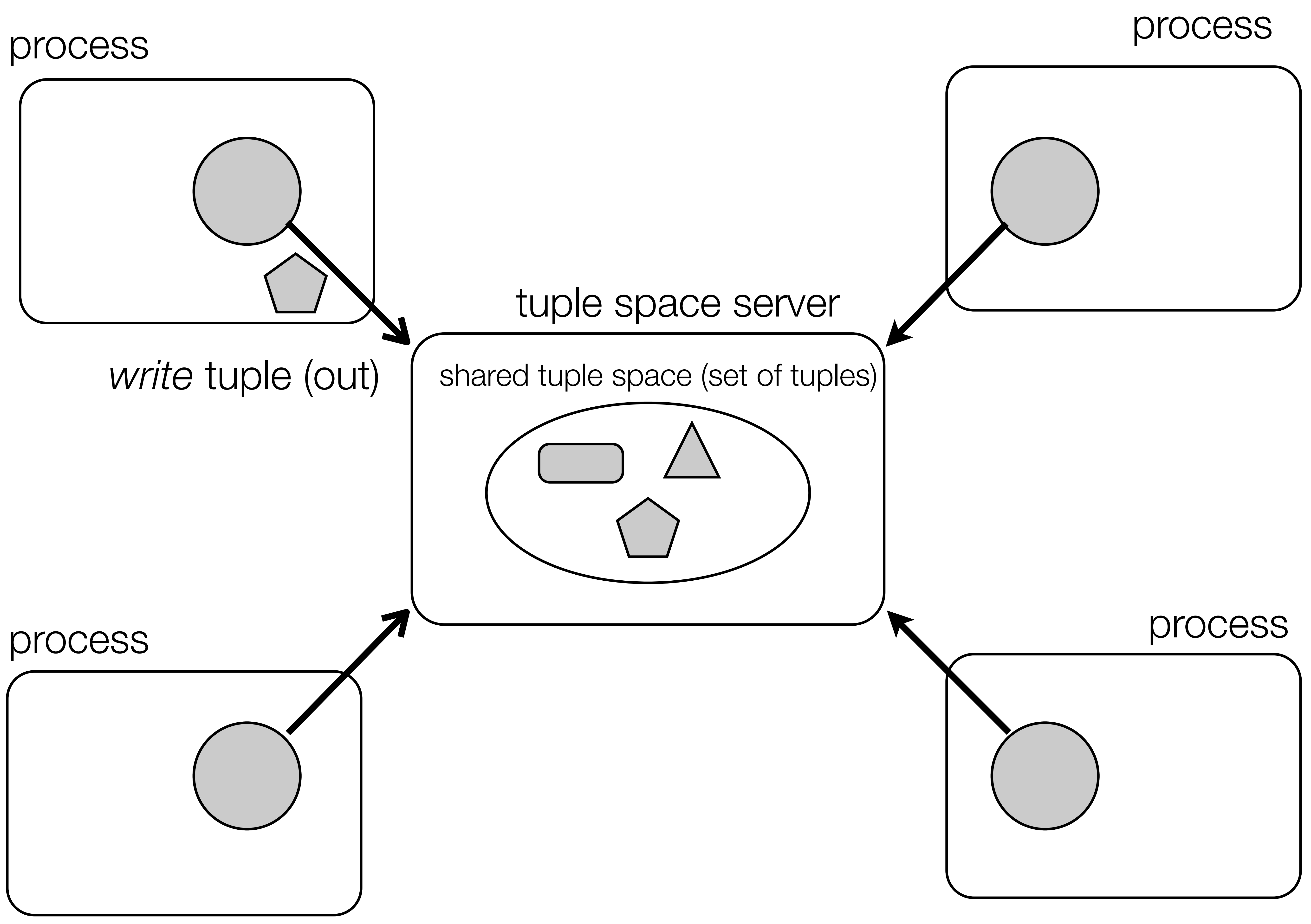
Tuple Spaces



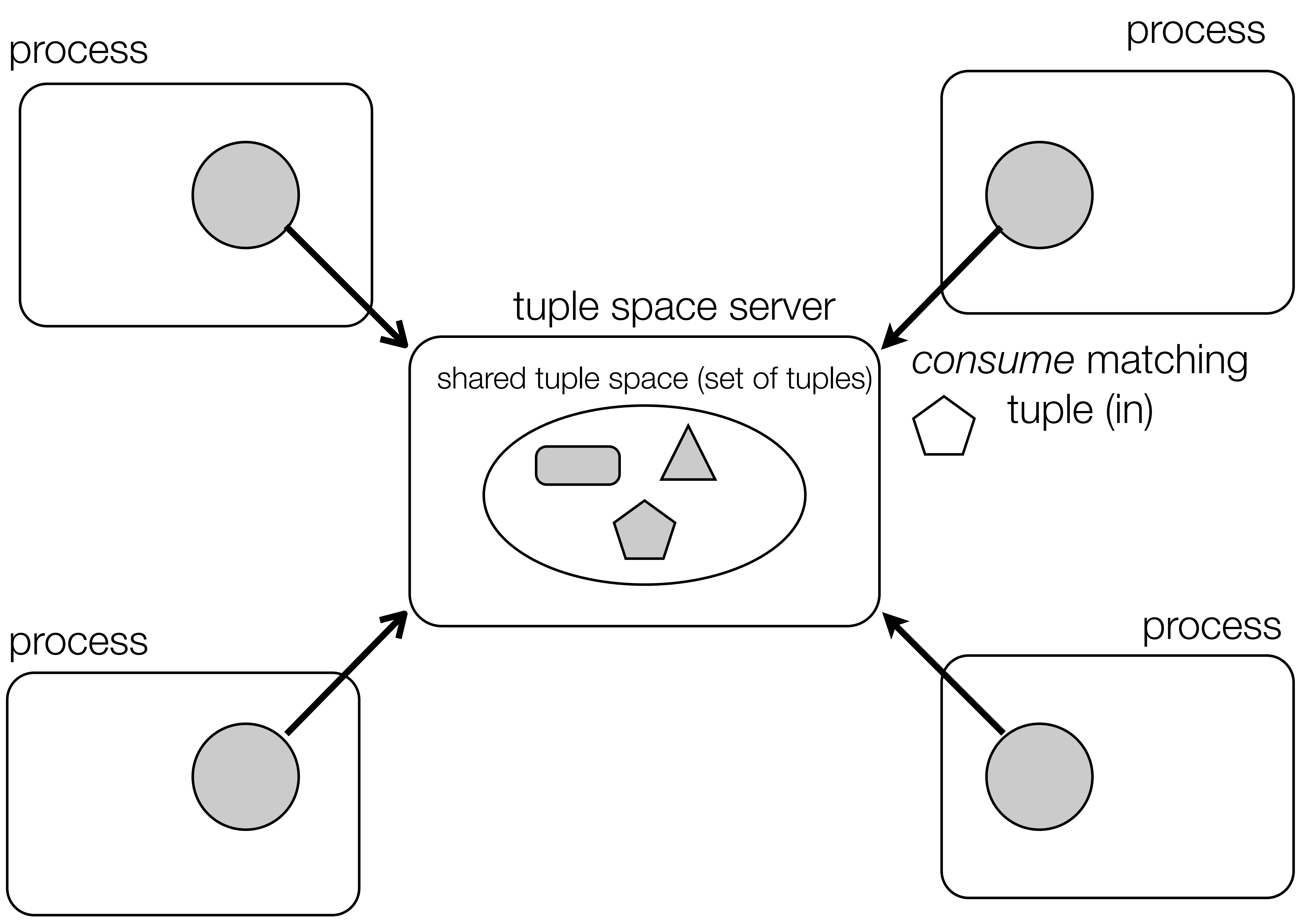
Tuple Spaces



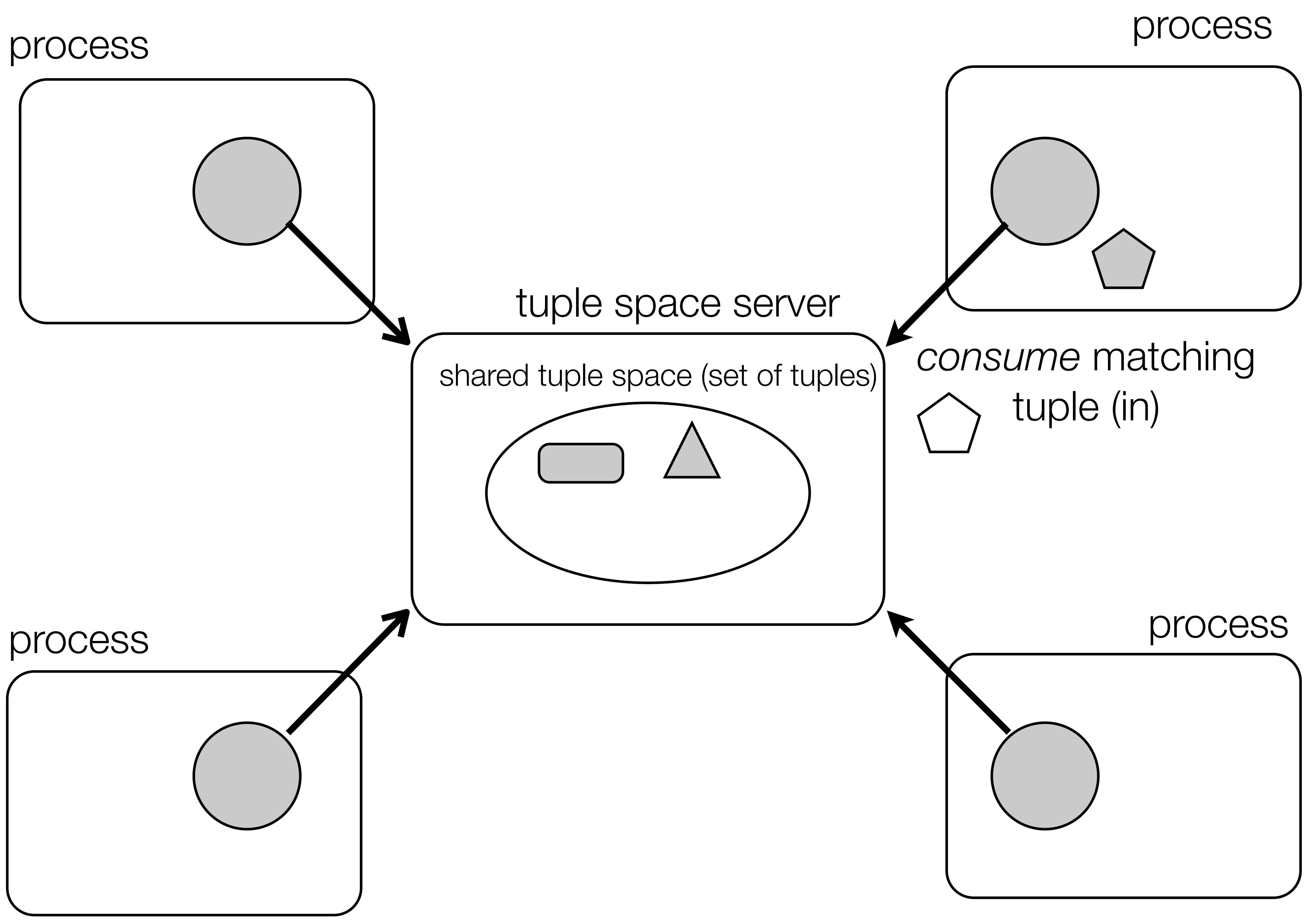
Tuple Spaces



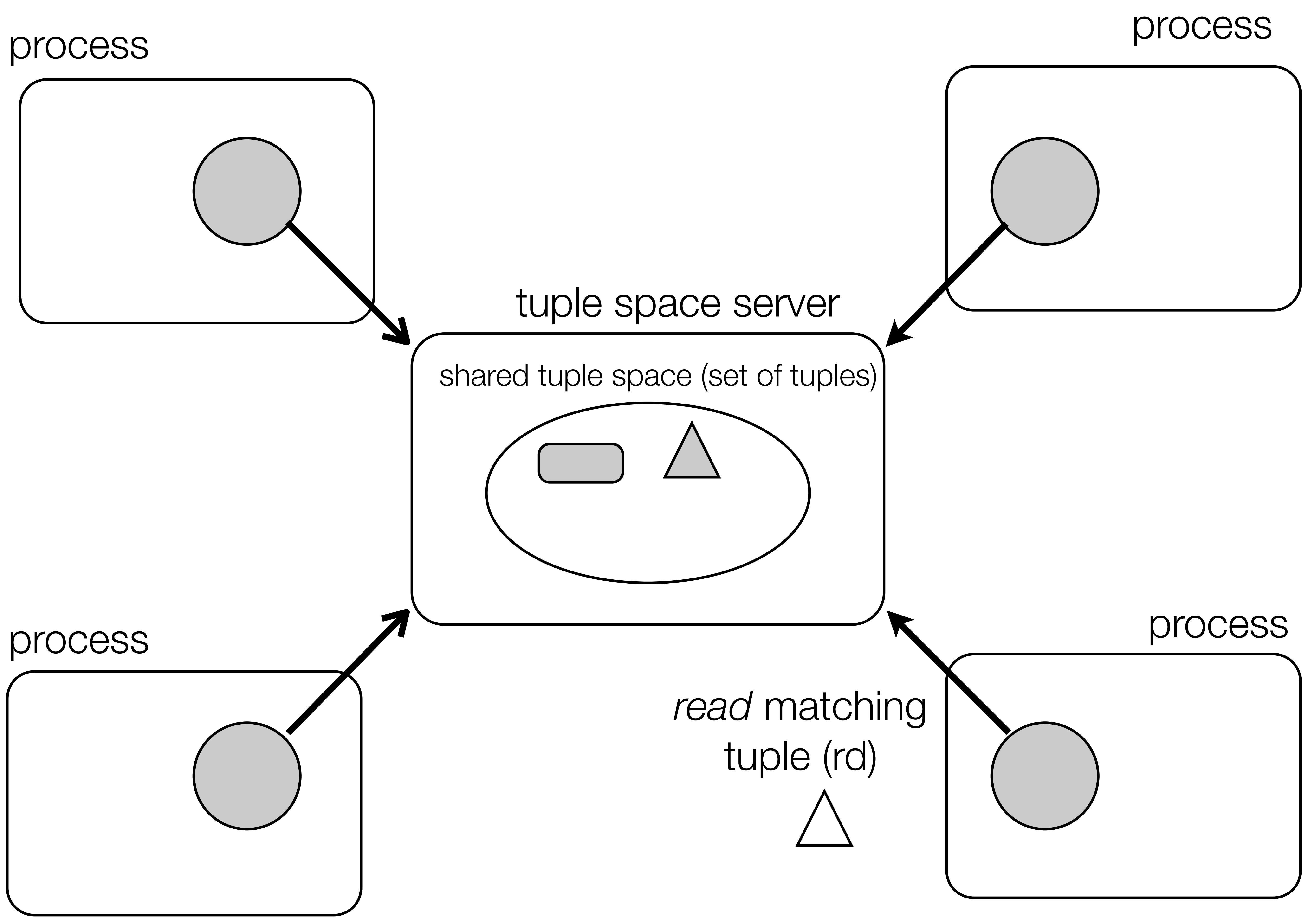
Tuple Spaces


process
process
tuple space server
shared tuple space (set of tuples)
consume matching tuple (in)
process
process

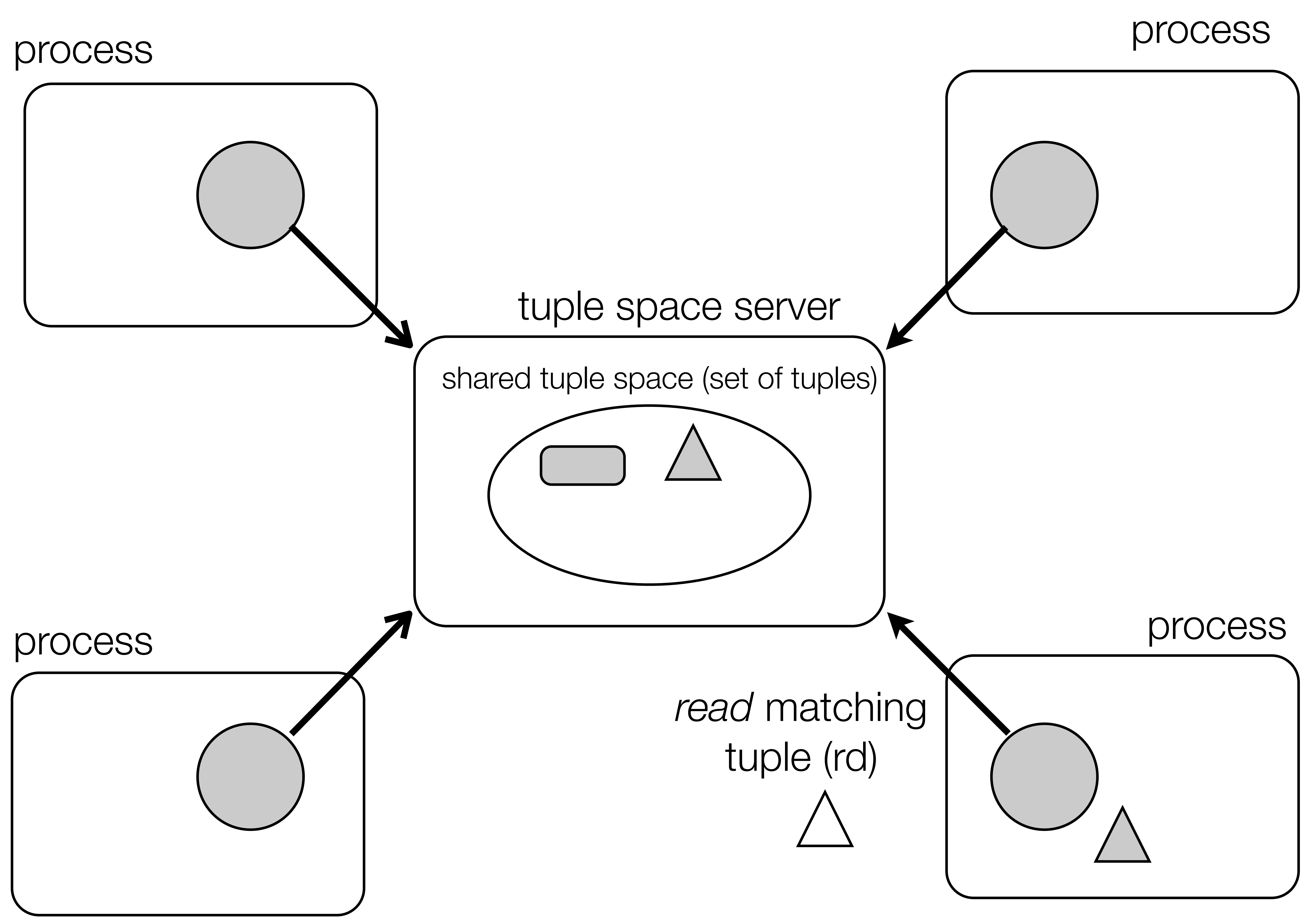
Tuple Spaces


process
process
tuple space server
shared tuple space (set of tuples)
consume matching tuple (in)
process
process

Tuple Spaces


process
process
tuple space server
shared tuple space (set of tuples)
process
process
read matching tuple (rd)

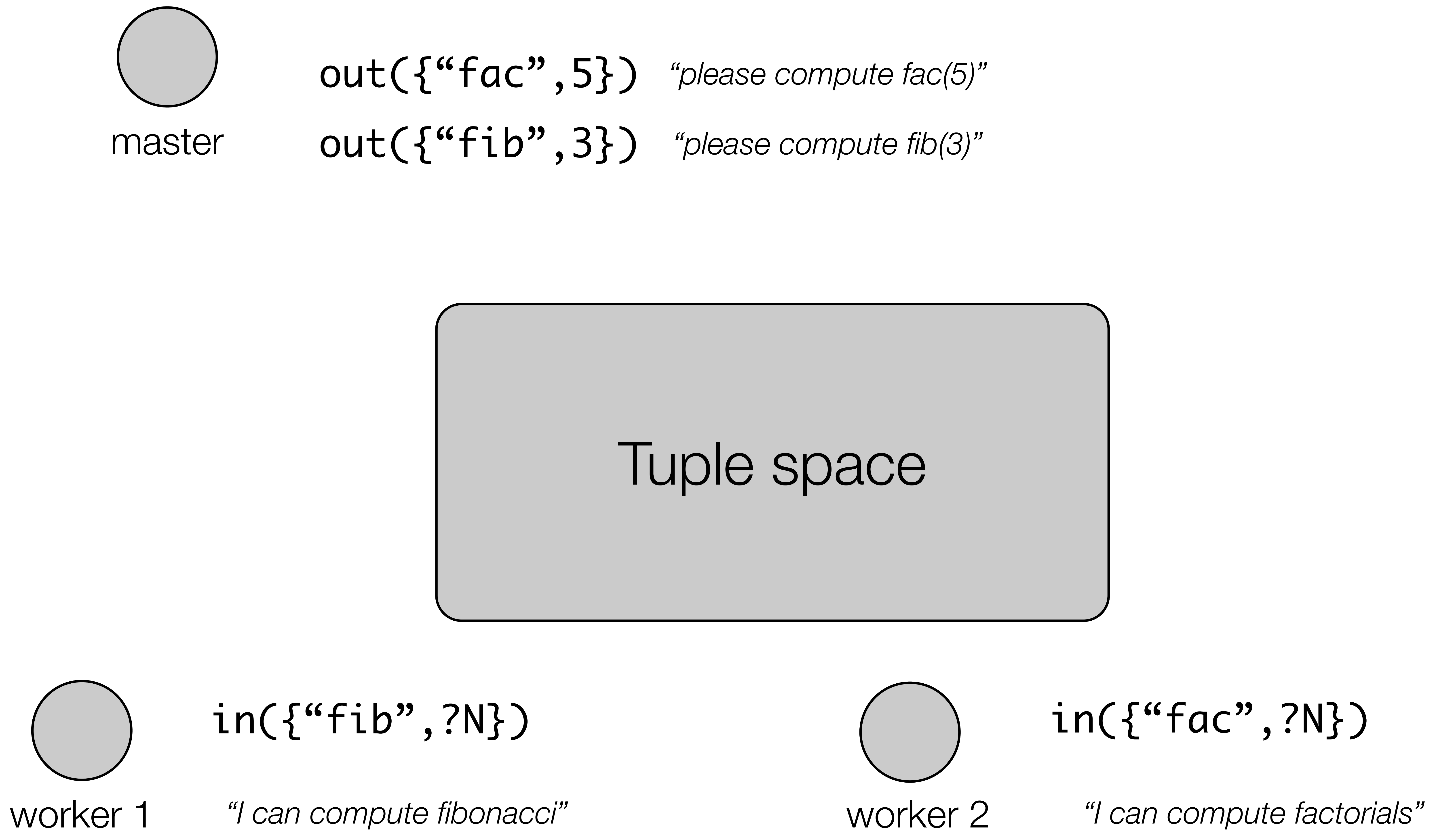
Tuple Spaces


process
process
tuple space server
shared tuple space (set of tuples)
process
process
read matching tuple (rd)

Tuple Spaces: Interface

- `out(t)`: add t to the tuple space
- `in(t):` remove t from the tuple space
    - If multiple processes simultaneously try to remove the same tuple, **only one** will succeed
- `rd(t)`: read t from the tuple space (t remains in the tuple space)
    - If multiple processes simultaneously try to read the same tuple, they will **all** receive a copy
- `in(t)` and `rd(t)` are blocking operations: process waits until a matching tuple is found
- There exist non-blocking variants, `inp(t)` and `rdp(t)` that only *probe* the tuple space, and either return a tuple or null if no matching tuple exists

Tuple Spaces Example



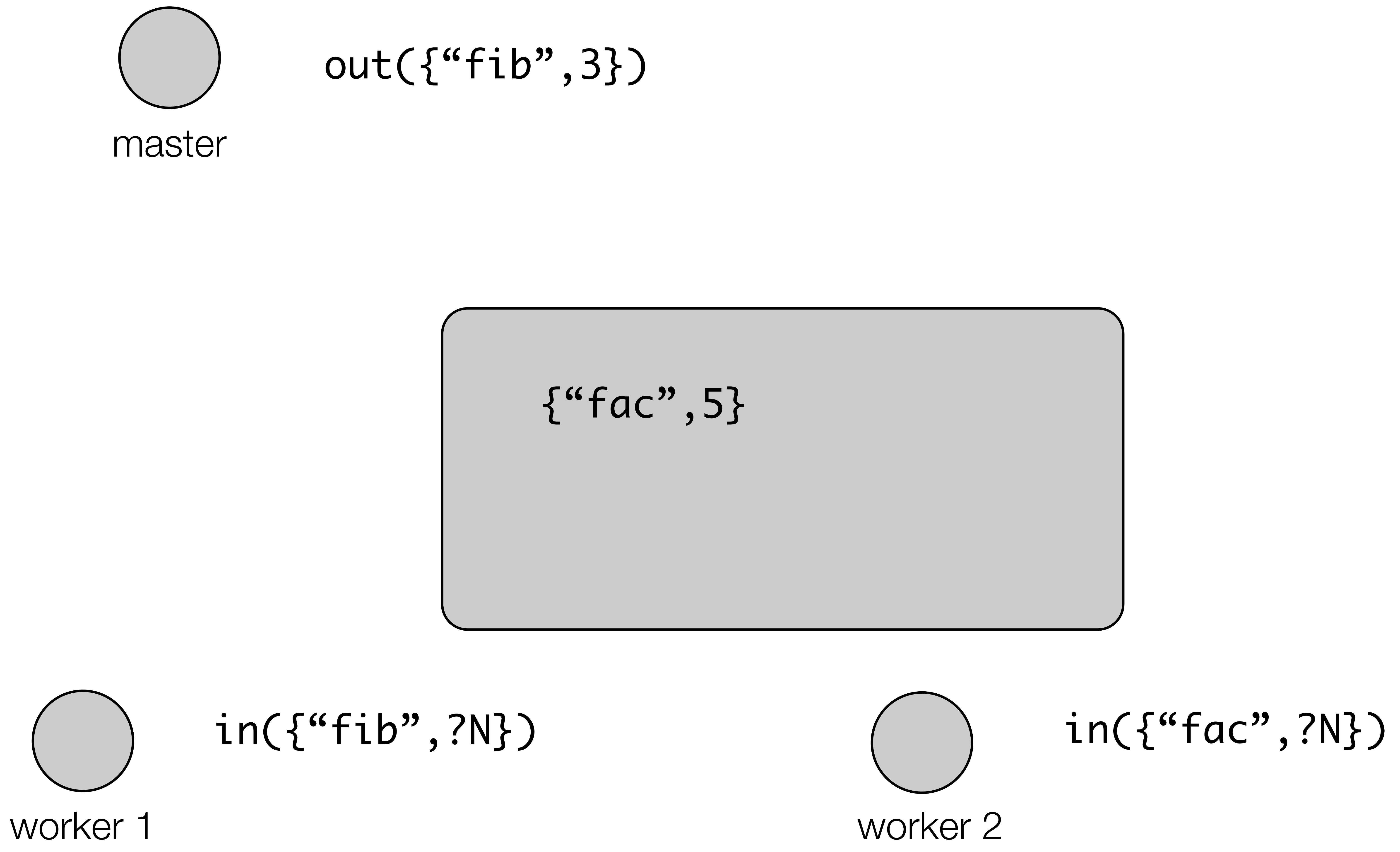
```
out({“fac”,5})
```
“please compute fac(5)”

```
out({“fib”,3})
```
“please compute fib(3)”

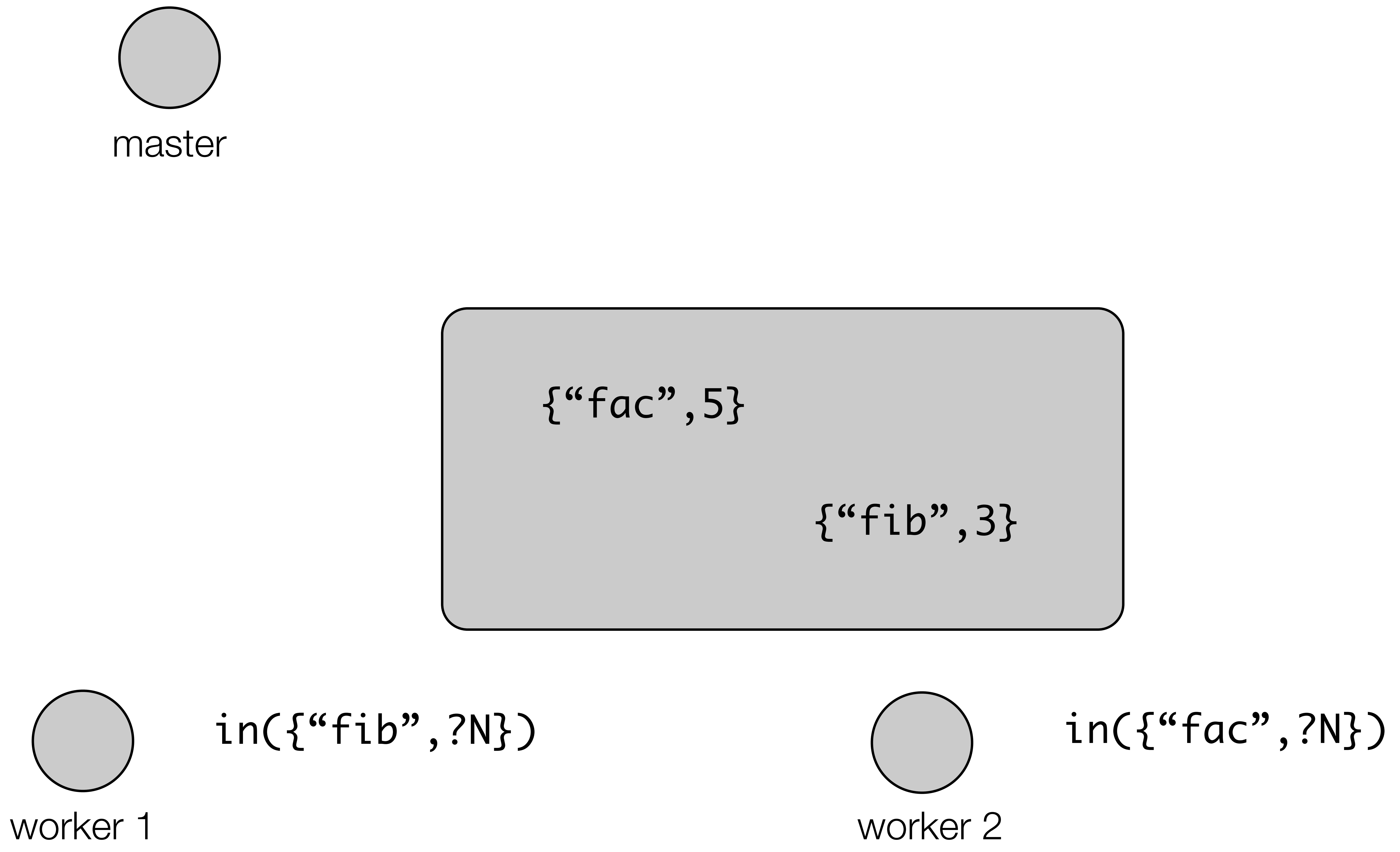
```
in({“fib”,?N})
```
“I can compute fibonacci”

```
in({“fac”,?N})
```
“I can compute factorials”

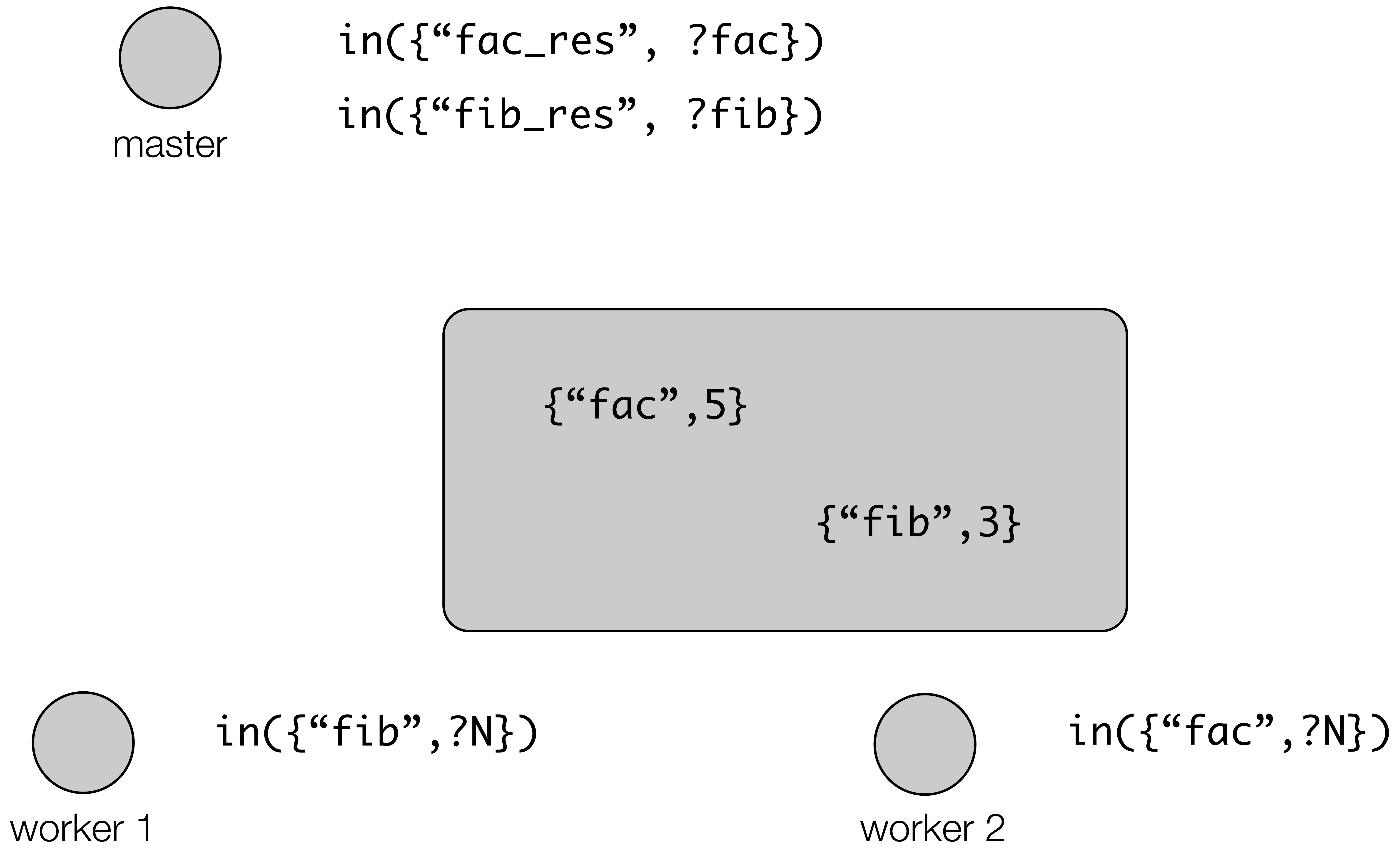
Tuple Spaces Example



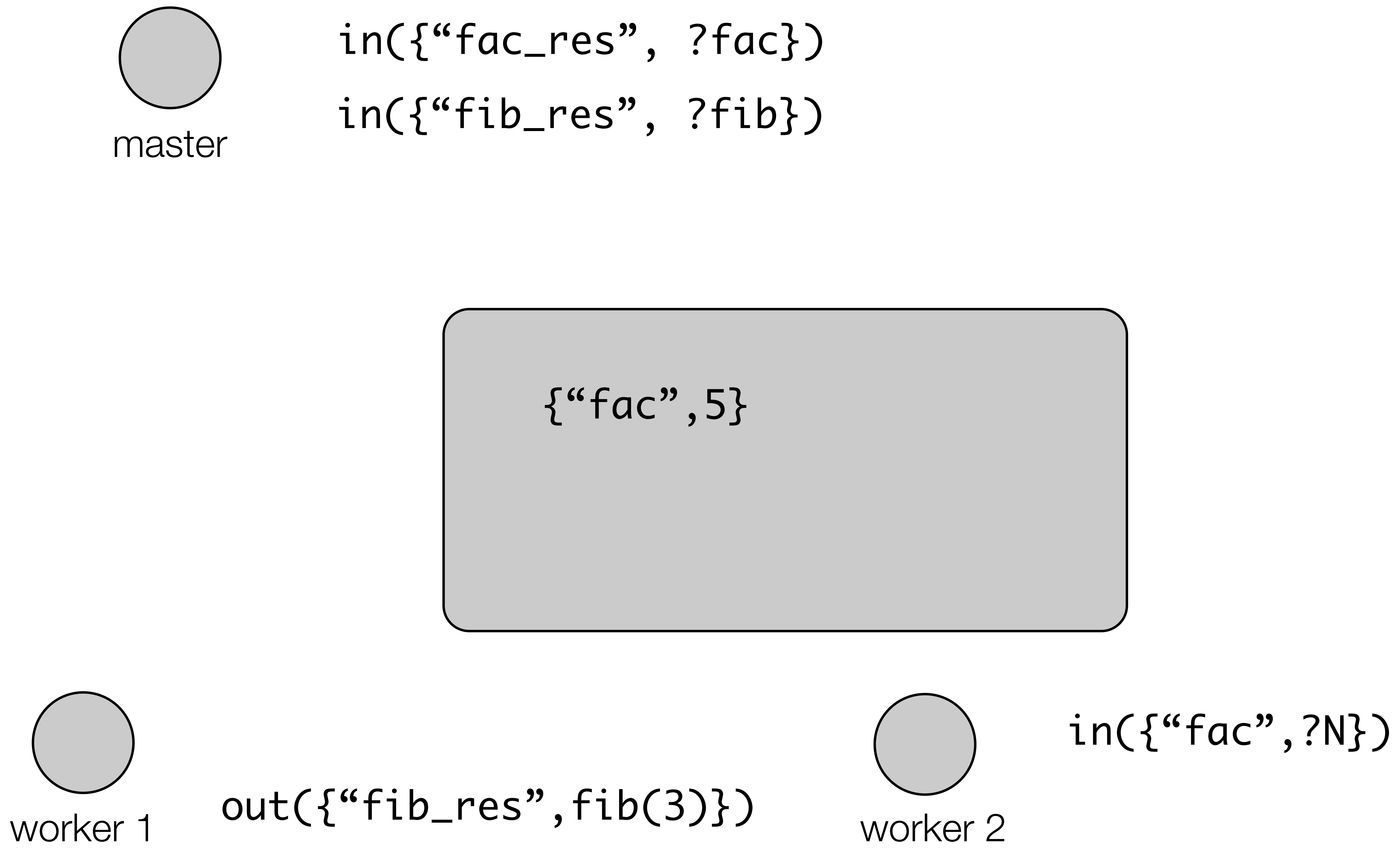
Tuple Spaces Example



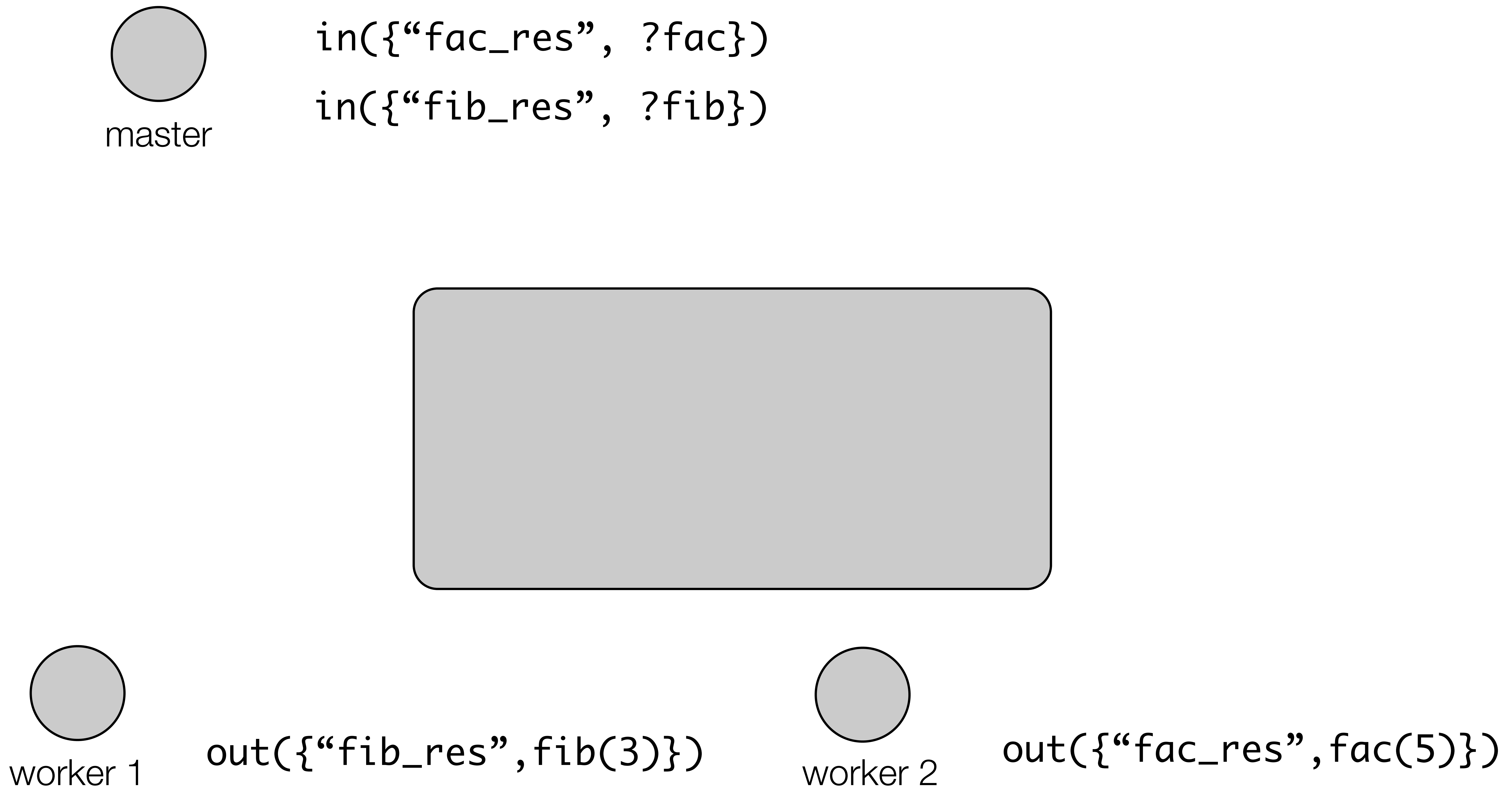
Tuple Spaces Example



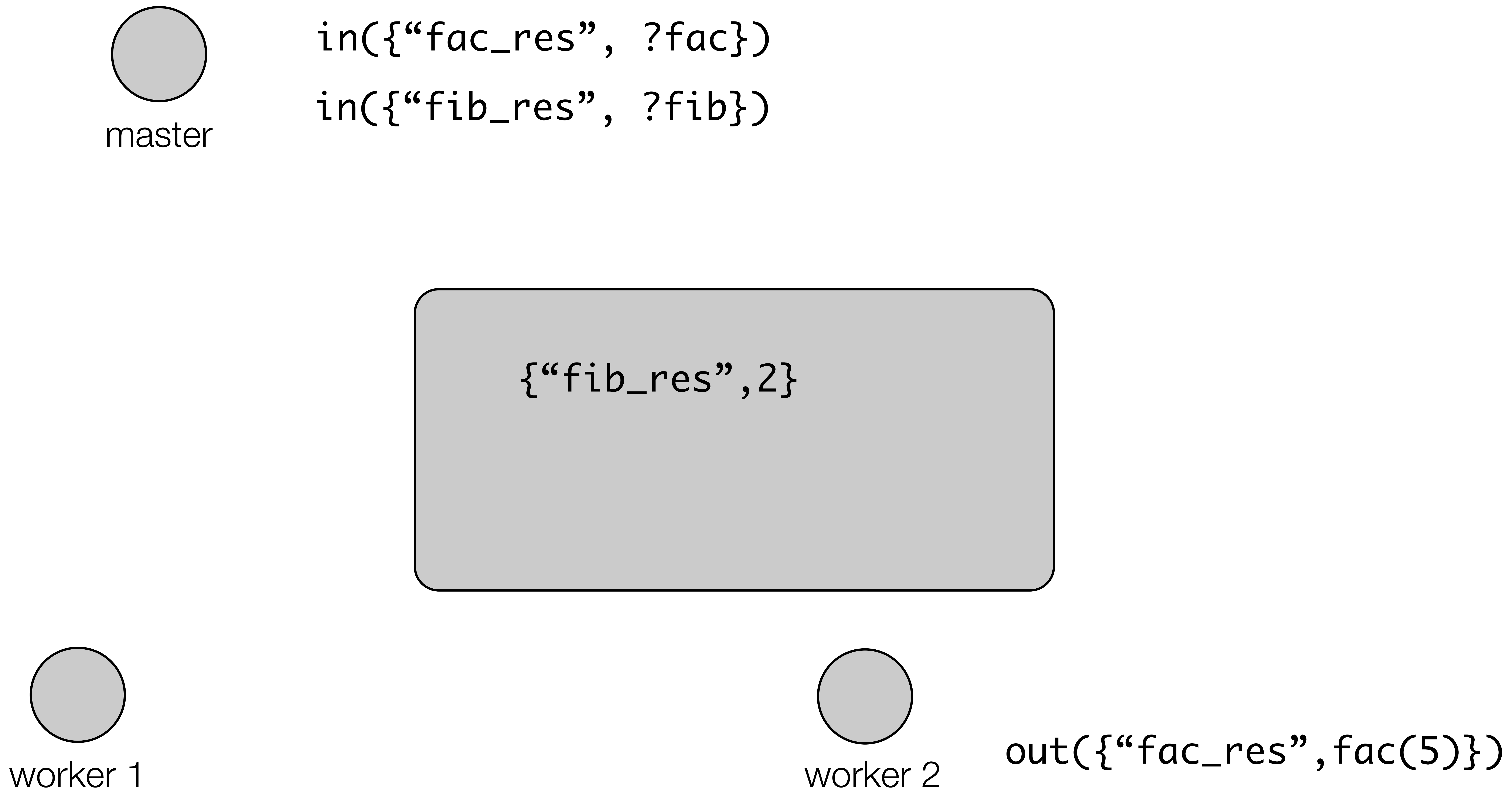
Tuple Spaces Example



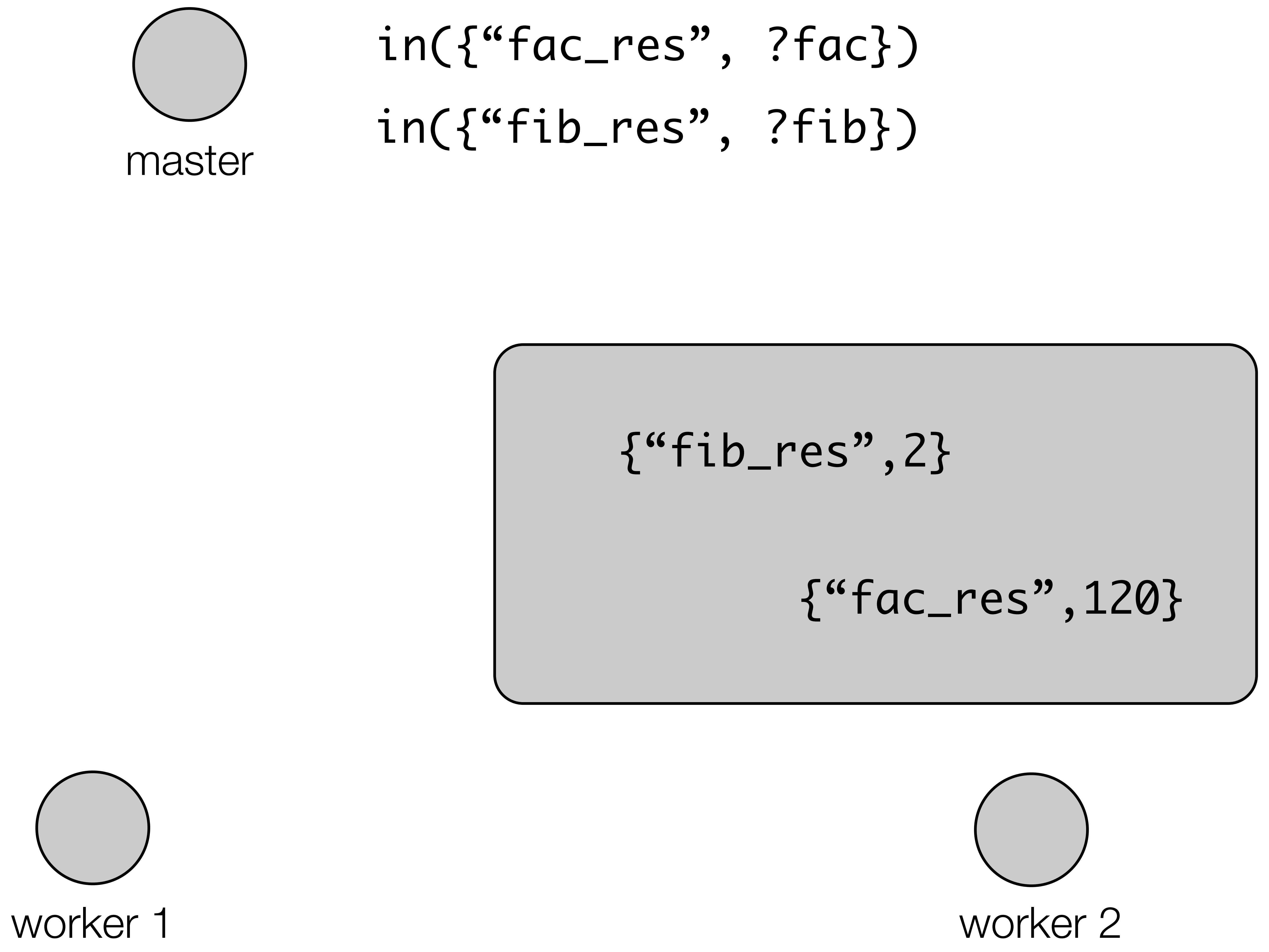
Tuple Spaces Example



Tuple Spaces Example



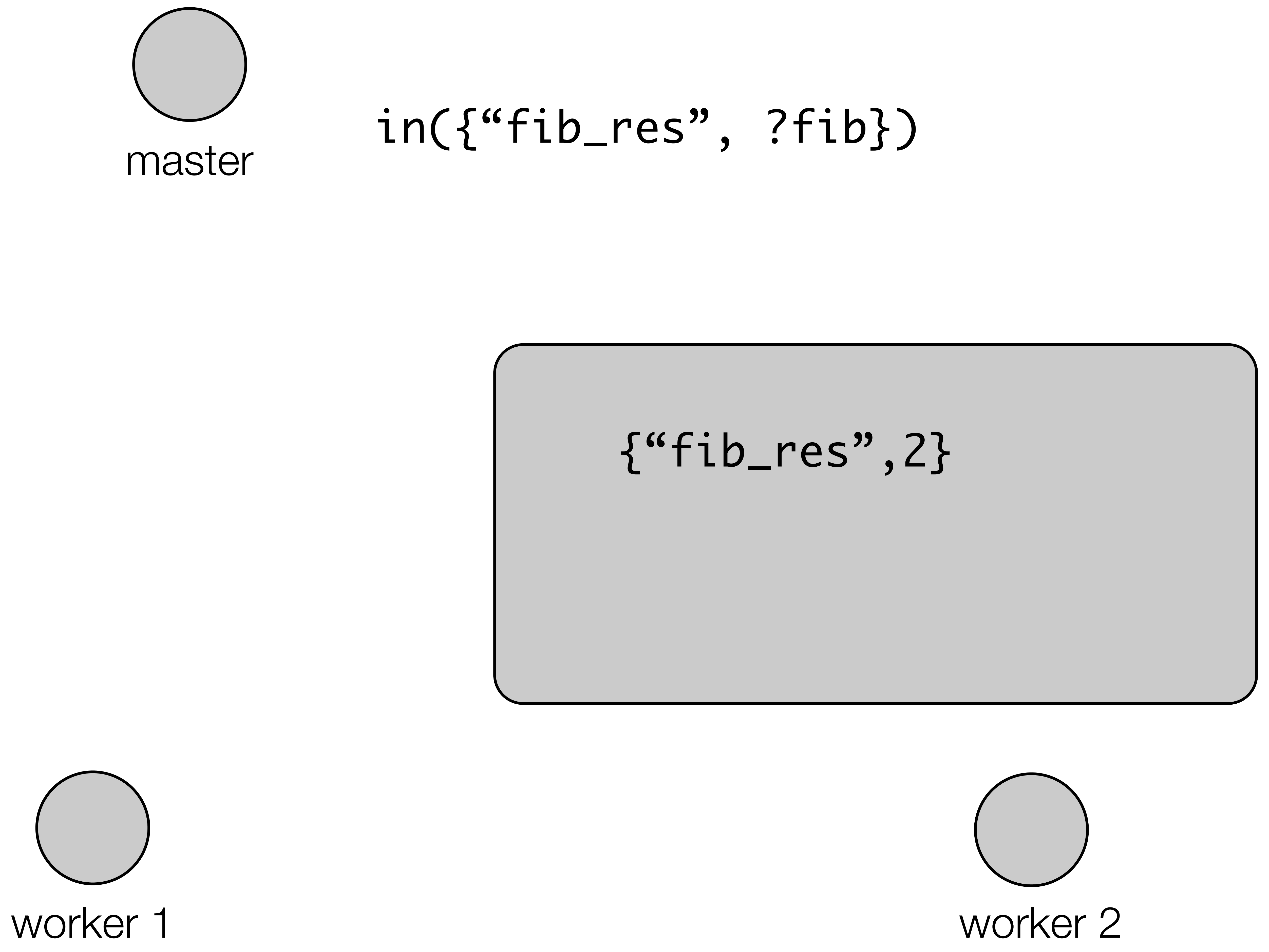
Tuple Spaces Example



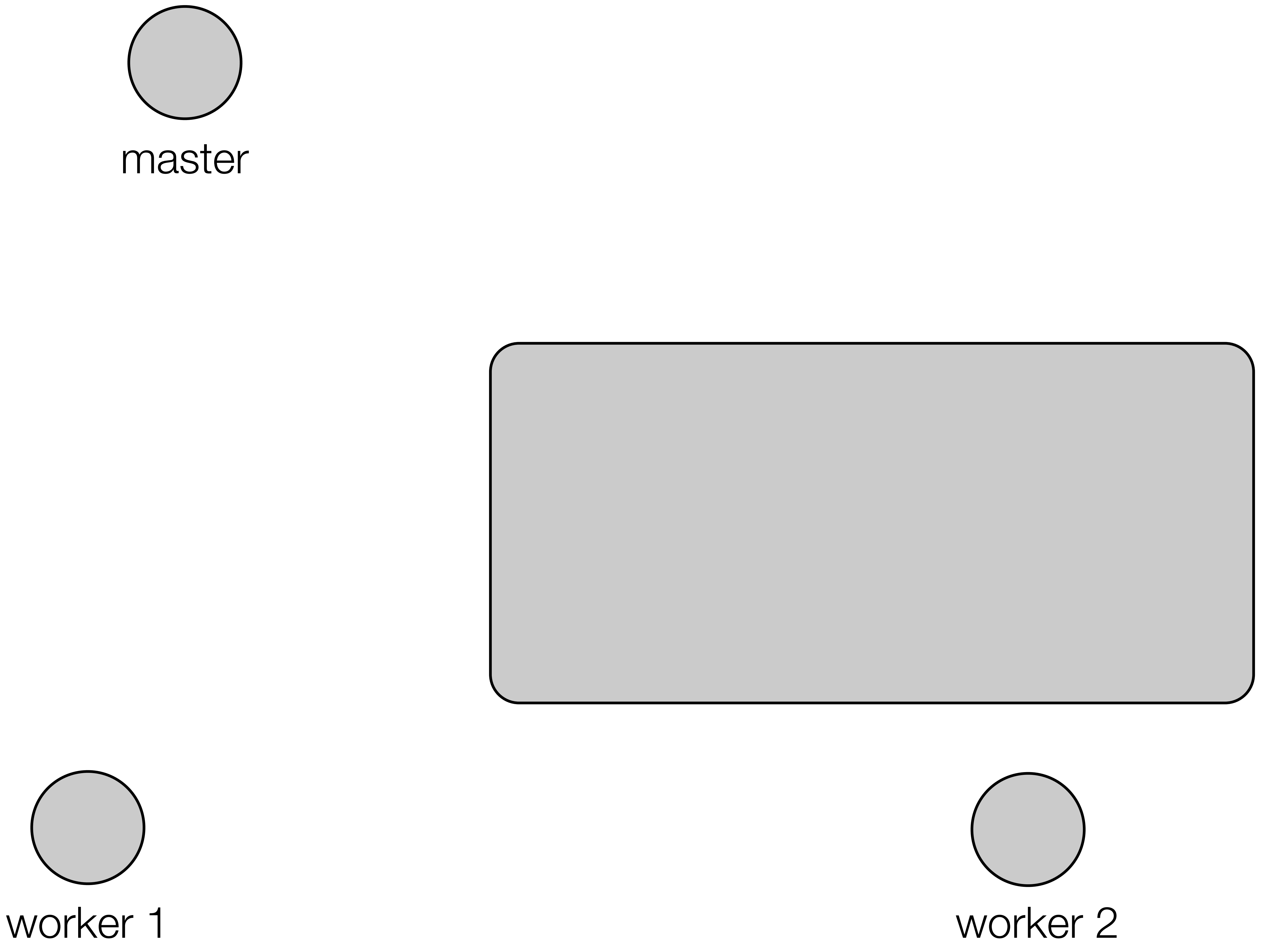
```
in({“fac_res”, ?fac})
in({“fib_res”, ?fib})
```

```
{“fib_res”,2}
{“fac_res”,120}
```

Tuple Spaces Example



Tuple Spaces Example



Tuple Spaces: summary

- Time-decoupled: the tuple space stores messages on behalf of processes so they don’t all need to be online at the same time.
- Space-decoupled: processes only need to know the shared tuple space, not the identity of other connected processes.
- Supports many-to-many communication
- Unlike all the previous communication paradigms, which are ***message-oriented***, tuple spaces are ***state-oriented*** (reading and writing values rather than exchanging messages)
- The API of tuple spaces is tremendously **flexible**: processes can act as consumers, producers, publishers or subscribers depending on whether they read, write or consume tuples.
- The flexibility of tuple spaces is also their **weakness**: they are difficult to implement efficiently and difficult to scale to large numbers of processes.
- While tuple spaces are **not widely used** in practice, they are similar to key-value stores which are very widely used

Communication patterns: recap

- Request-reply
- Event Notification
- Pipeline (push-pull)

Direct communication (time and space-coupled)

- Publish-subscribe
- Message queue
- Tuple space

Indirect communication (time and space de-coupled)

Communication patterns: summary

	Request-reply	**Event Notification**	**Pipeline (Push-Pull)**	**Publish-subscribe**	**Message queue**	**Tuple space**
Time uncoupled? (Need not all be online at same time)	No	No	No	Yes (if messages stored by broker)	Yes	Yes
Space uncoupled? (Need not know address of other processes)	No (direct addressing)	No (direct addressing)	No (direct addressing)	Yes (indirect via topic name)	Yes (indirect via queue name)	Yes (indirect via tuple structure)
Communication pattern	1-to-1	1-to-many	many-to-many	many-to-many	many-to-many	many-to-many
Use cases	Basic client-server requests	Real-time information dissemination	Load balancing of work (tasks)	Enterprise application integration (EAI)	EAI, transaction processing	Load balancing of work, sharing data updates
Scalable?	Server is bottleneck (or use load balancer)	Possible (hierarchical subscriptions)	Limited, M x W direct connections	Possible (federated brokers)	Possible (federated brokers)	Limited (no easy way to split a tuple space across machines)
Communication is	Message-oriented	Message-oriented	Message-oriented	Message-oriented	Message-oriented	State-oriented
Associative naming? (communicate based on the value of data)	No	No	No	No for topic-based, yes for content-based	No	Yes

Communication Paradigms: next lectures

- Three major paradigms:
  - 1. **Inter-process communication** (reading/writing a byte stream shared between two processes)
    - Usually based on OS network programming APIs (e.g. UNIX sockets)
  - 2. **Remote invocation** (basic request/response interaction between two processes)
    - Remote procedure call (in procedural languages) or remote method invocation (in object-oriented languages)
  - 3. **Indirect Communication** (communication with a known or unknown group of processes)
    - Often by registering “Observers” or “Listeners” or “Callbacks” on an event source

Questions?

- Don’t forget to check out the code examples on Github:
- https://github.com/tvcutsem/distributed-systems
- See the folders in `communication_patterns/`