indirect communication

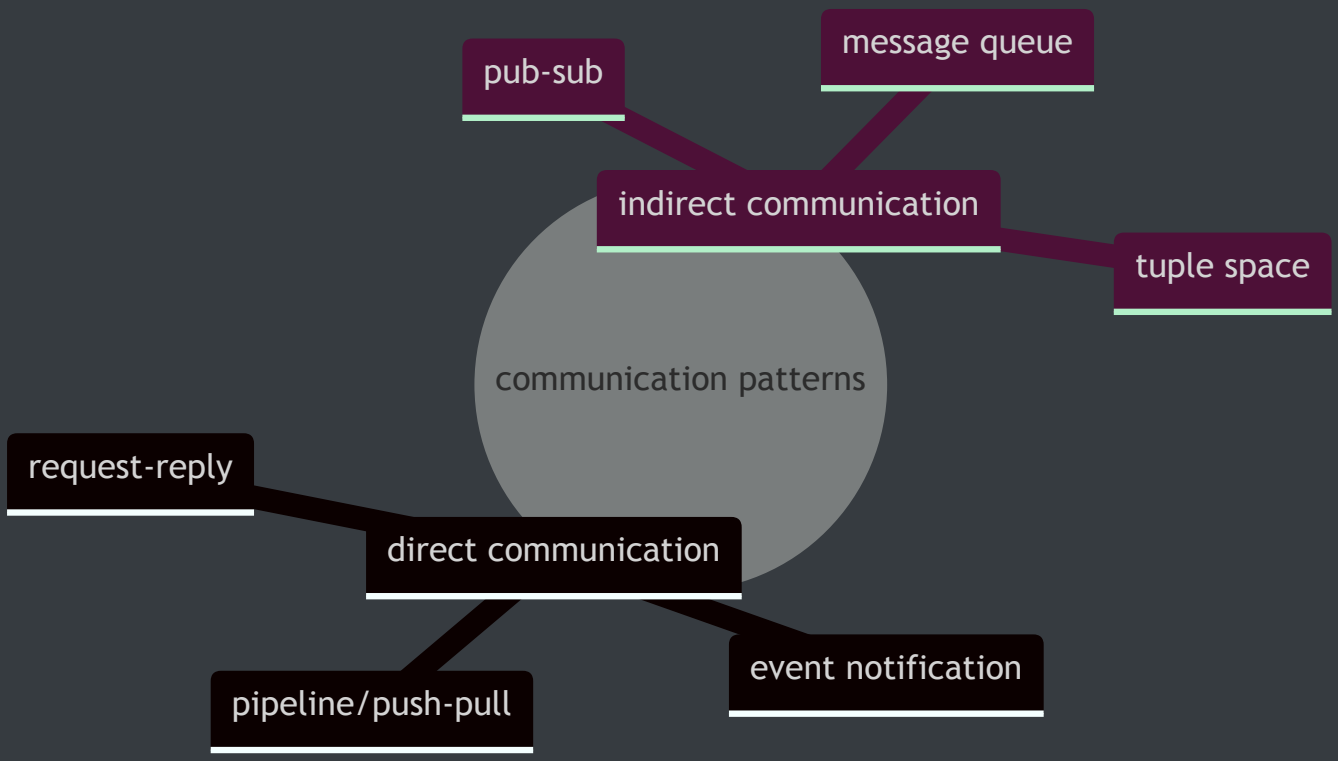
code examples of this chapter:

zeroMQ - used for direct communication

rabbitMQ - used for indirect communication

motivation of indirect comm.: decoupling in TIME & SPACE

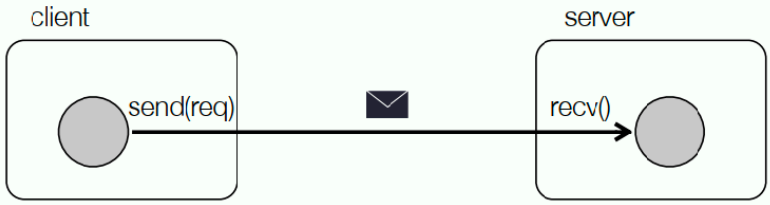
- communication parties need not be online at the same time
- sender does not need to know the receivers' address / identity



communication patterns

- direct -- time & space coupled
    1. request-reply -- one-to-one

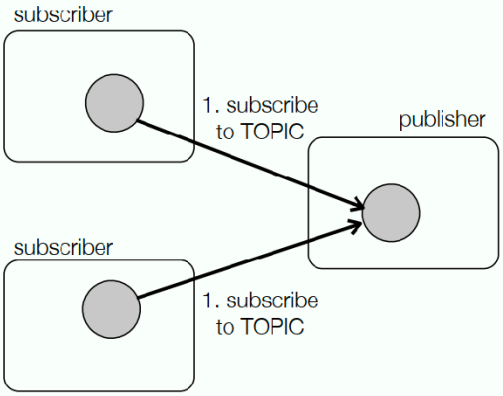
- Client directly connects to server, server accepts requests from any client
- Server sends response, client blocks until response is received



- often synchronous (e.g. RPC) but can be asynchronous

2. event notification -- one-to-many

- 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

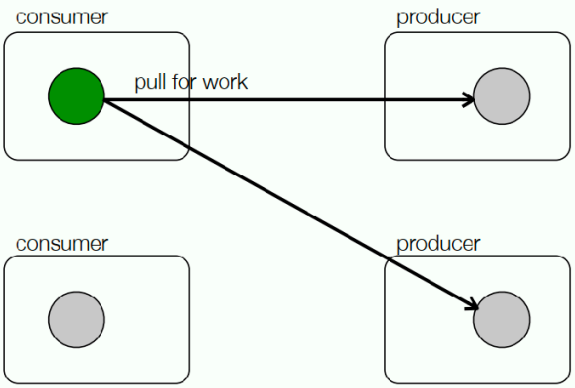


ØMQ
/event_notification

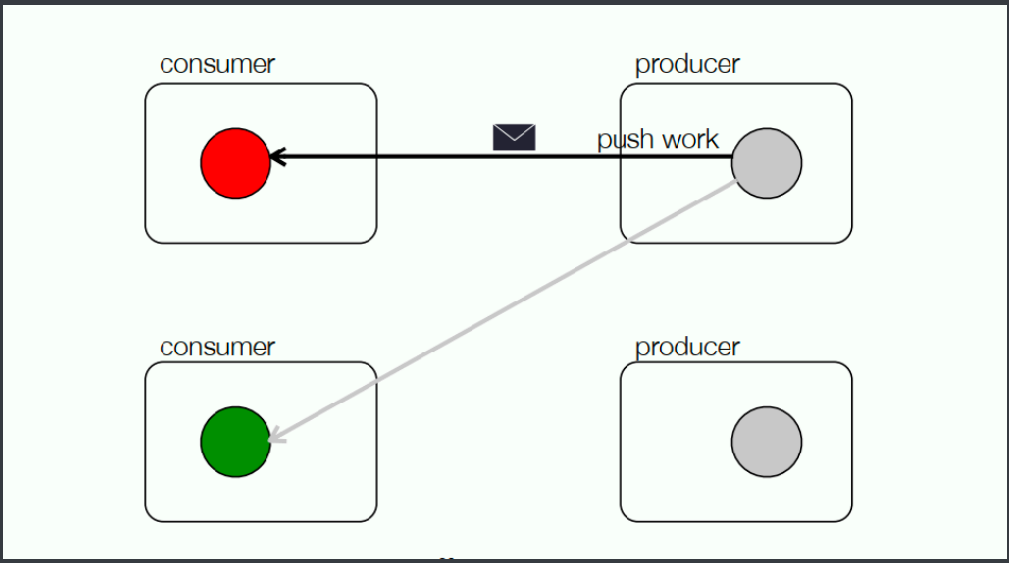
- topic name acts as filter
- subscribers listen for events, publisher "fire-and-forget"
- practical for real-time, e.g. IoT sensor readings

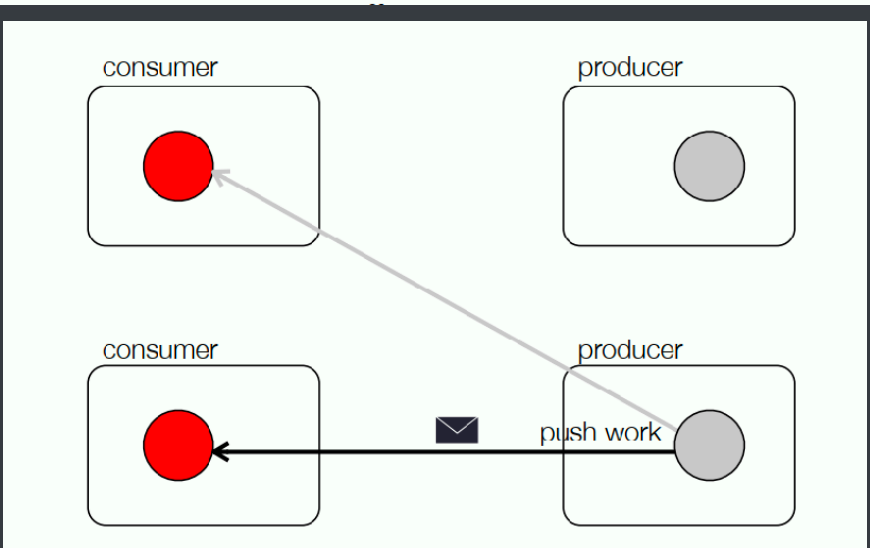
3. pipeline (push-pull) -- many-to-many

- 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





- synchronous:

  consumers block if no producer pushes work,

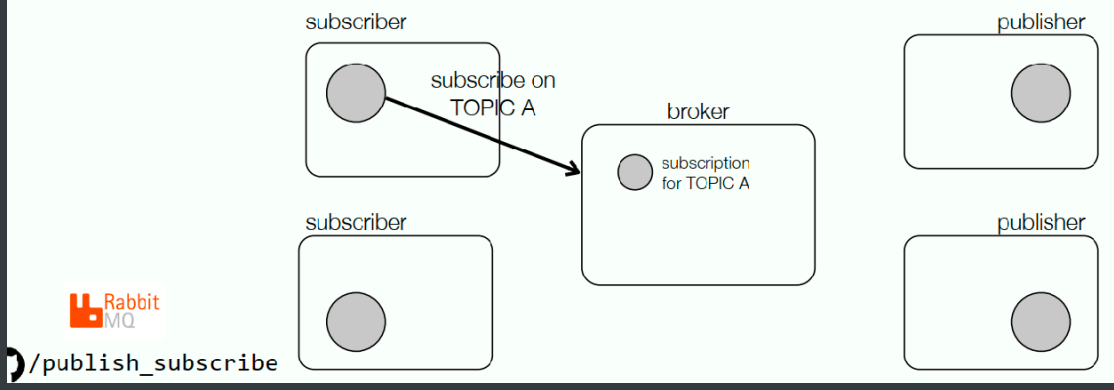
  producers block if no consumer pulls for work
- usually one-way

  typically consumer does not get the result back to the producer
- useful for load-balancing, fair queueing strategy needed

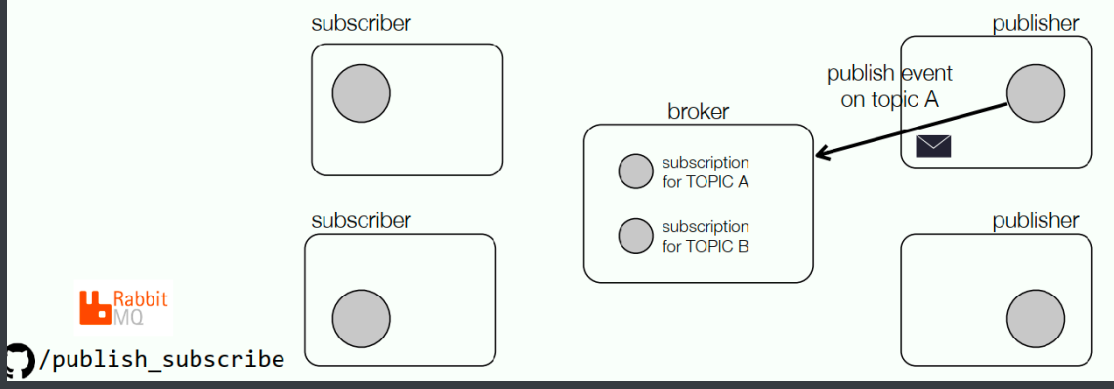
- indirect

1. pub/sub -- many-to-many

- 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.



- 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)



- broker not necessarily on the same machine as either the publisher or the subscriber
- 2 approaches of matching subscriptions with event notifications

  topic-based

  use a shared topic name

  content-based

  a event consists of attribute-value pairs,

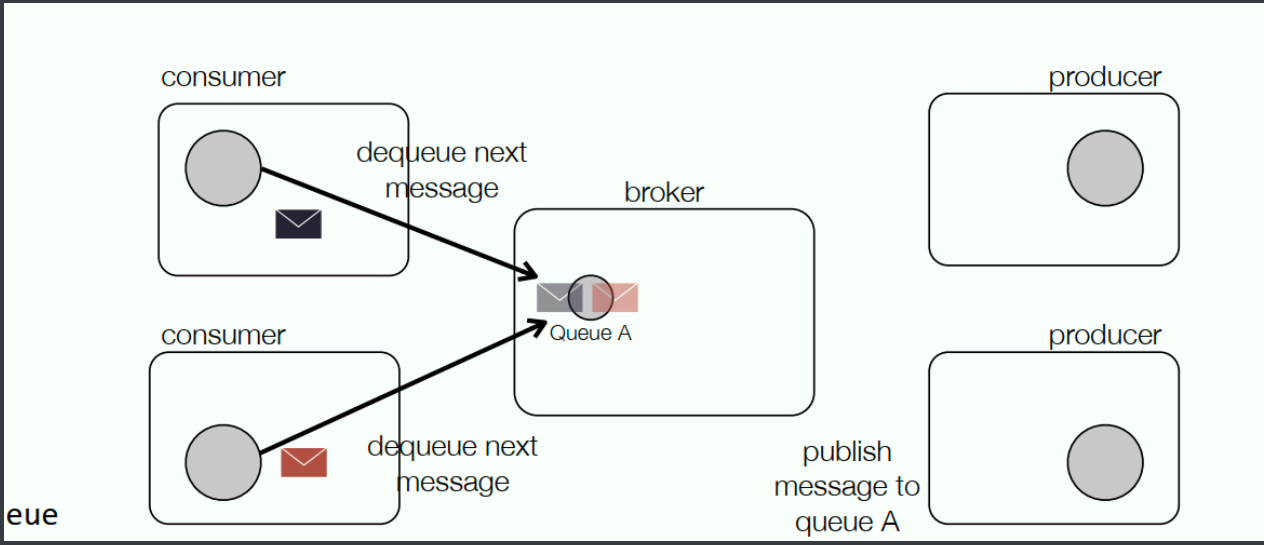
  (e.g. `{“city”=“Leuven”, “weather”=“sunny”, “temp”=32, “date”=“01-08-2023”}`)

  subscription can be predicates on attribute values,

  e.g. “city”=* AND “temp” > 28 AND “date” OLDER THAN “01-01-2023”
- not time-coupled (but if the subscriber does not create a queue, then it mush remain connected to the broker)

2. message queue (many-to-many)

   similar to pipeline (push-pull), but with queue held by the broker



   - multiple consumers can register to hte same queue
   - **message queueing vs. pub-sub**

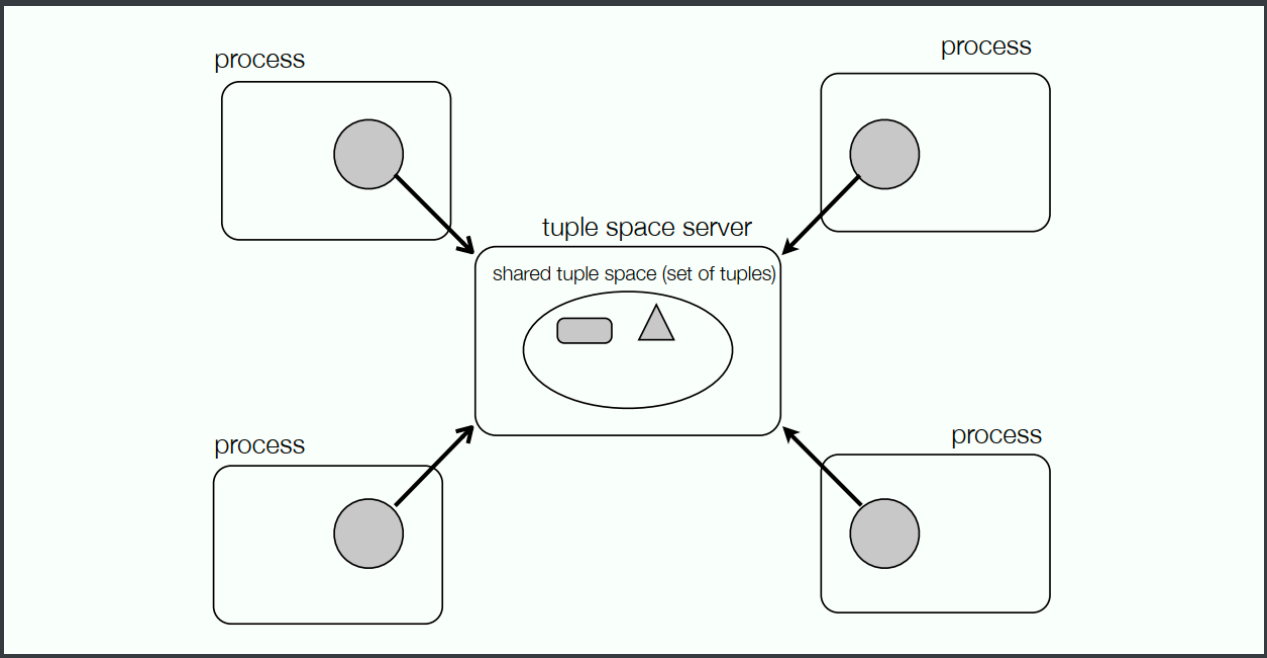
     in pub-sub subscribers are independent !!!

     each message is sent to all matching subscribers, and each subscriber can have its own queue;
   - **message queueing vs. pipeline (push-pull)**

     producers don't have to block to wait for consumers' availability

3. tuple space

   a logically shared memory space



- what is a tuple?

  ordered sequence of typed data values:

  ```
  {"john", 42}
  ```

  could be read based on a template / wildcards:

  ```
  {"john", ?age}
  ```

- operations:

  `out(t)` : add t to tuple space

  `in(t)` : consume. or say, remove t from tuple space

  `rd(t)` : read only, t remains in the tuple space

  Note:

  `in` and `rd` are blocking operations, process waits until a matching tuple is found

  to make it non-blocking, we could use `intp(t)` / `rdp(t)` , which returns either a tuple or null if no matching

  (*p stands for probing*)

- **state-oriented**, rather than message-oriented
- flexible: process can be consumer/producer/publisher/...

  but difficult to efficiently implement / scale

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

AMQP: advanced message queueing protocol

- data structures: exchanges, queues
- events can be sent:

  to an exchange (then deliver to all queues with matching topic),

  or

  directly to a queue