Sistemas Distribuidos I
(75.74)

Patrones de Comunicación

Request-Reply, Publisher-Subscriber, Pipelines y DAG

Docentes

- Pablo D. Roca
- Ezequiel Torres Feyuk
- Guido Albarello
- Ana Czarnitzki
- Cristian Raña

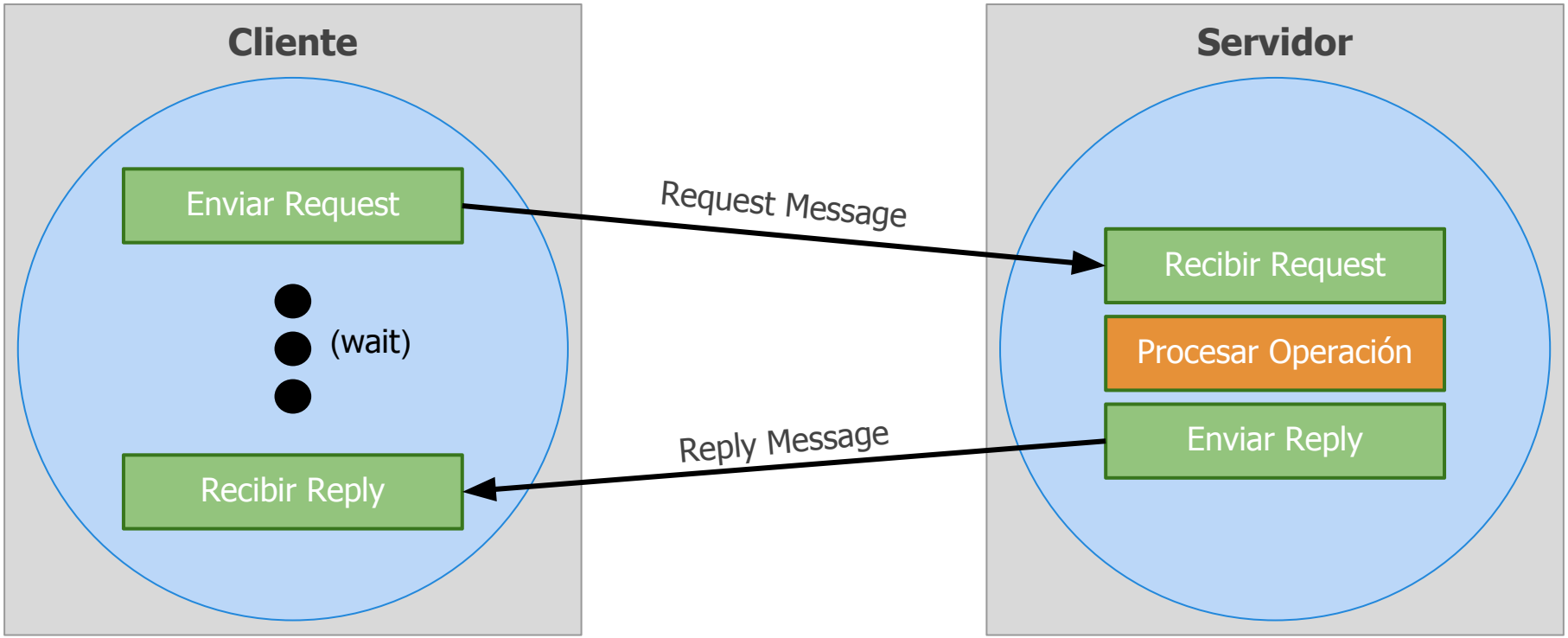
Agenda

- **Request-Reply**
- Publisher-Subscriber
- Pipelines y DAGs

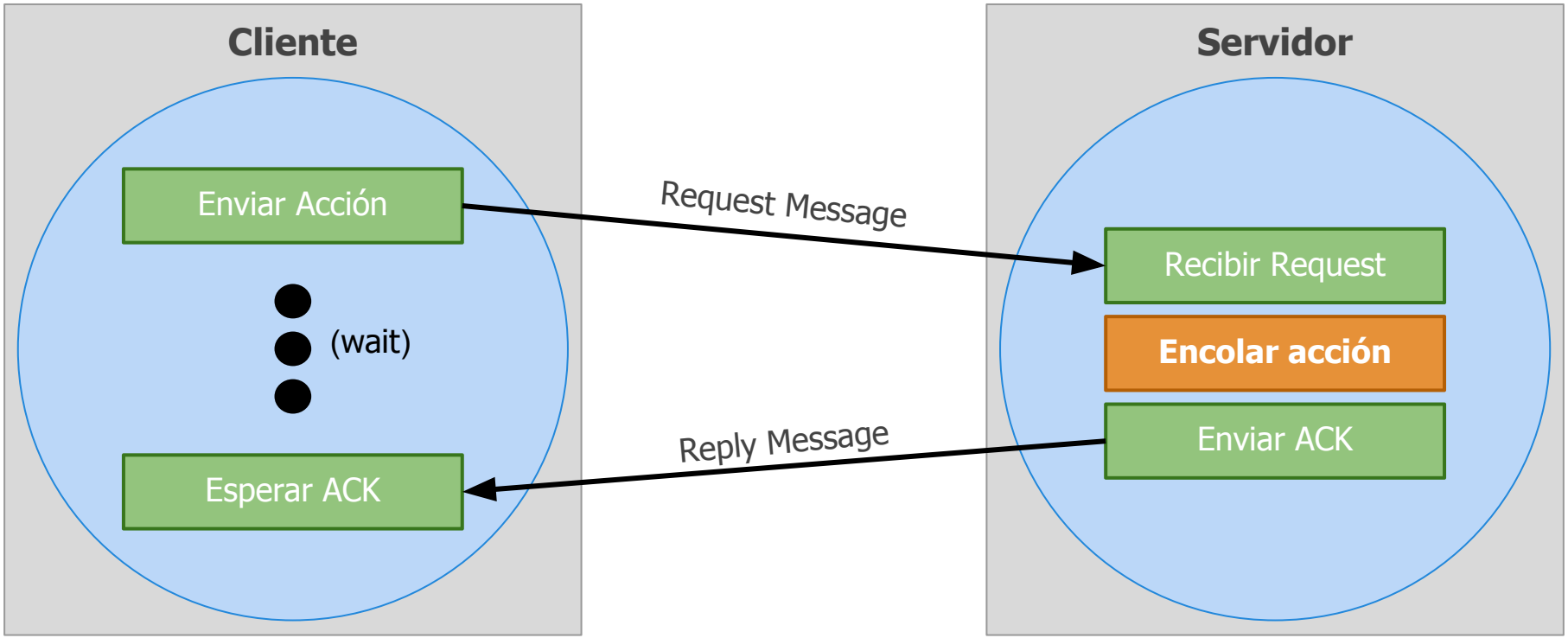
Request-Reply | Introducción

- Protocolo utilizado en modelo Cliente-Servidor
- **Es sincrónico (bloqueante) por defecto**
  - Cliente envía Request Message
  - Servidor recibe Request, procesa mensaje y envía Reply
  - Cliente queda bloqueado hasta recibir Reply Message
- ACK triviales (el Reply message es un ACK)
- **Cómo implementamos una operación asincrónica?**
  - 2 Request-Reply sincrónicos son necesarios
  - Primero se envía operación a realizar
  - Luego se obtiene resultado de la operación

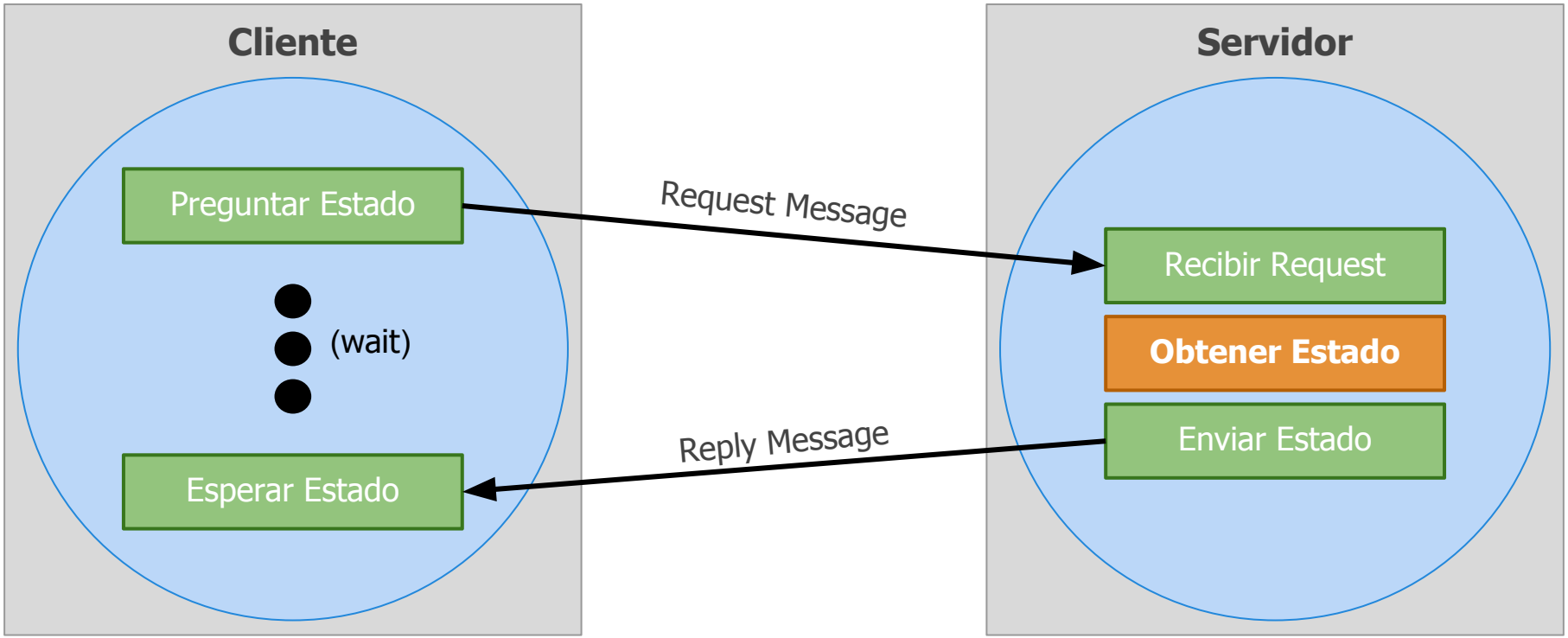
Request-Reply | Operación Sincrónica



Request-Reply | Operación Asincrónica (1° Parte)



Request-Reply | Operación Asincrónica (2° Parte)



Request-Reply | Estructura de mensajes

- Los siguientes campos suelen ser obligatorios
  - **messageID:** 0 = Request; 1 = Reply
  - **requestID:** Identifica unívocamente al mensaje enviado
    - Autoincremental
    - UUID
  - **operationID:** Identifica acción / operación a realizar
  - **Argumentos:** Atributos asociados a la acción / operación

Request-Reply | Tolerancia a Fallos

- **Cuánto se debe esperar por Reply?**
  - Timeouts con Retries
  - Algoritmos de Backoff + Jitter
  - Ejemplo Amazon API
- **Qué pasa si un Request o un Reply se pierde?**

Retry - Request	Filtro Duplicados	Retransmisión / Re-ejecución	¿Mensaje recibido?
No	No implementable	No implementable	***Maybe***
Si	No	Re-ejecución	***At Least Once***
Si	Si	Retransmisión	***Exactly Once***

Agenda

- Request-Reply
- **Publisher-Subscriber**
- Pipelines y DAGs

Publisher-Subscriber | Introducción

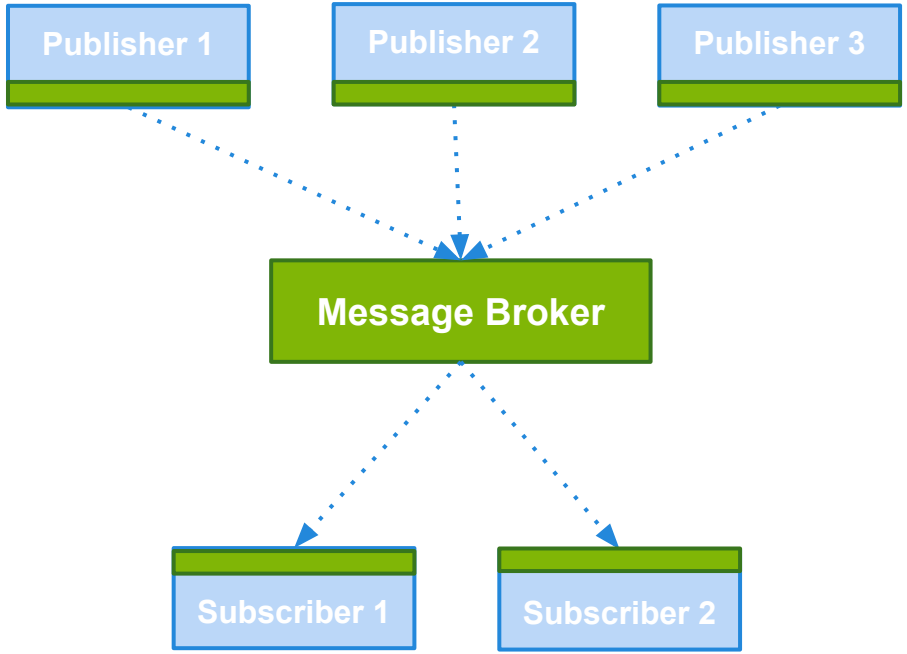
Modelo basado en comunicación por eventos entre productores y consumidores

- ***Publishers:*** también llamados *producers,* son los emisores. Componentes que tienen la posibilidad de generar algún elemento de interés.
- ***Subscribers:*** también llamados *consumers*, son los receptores. Esperan la aparición de algún evento de su propio interés sobre el cual efectuarán alguna acción.

Publisher-Subscriber | Arquitectura

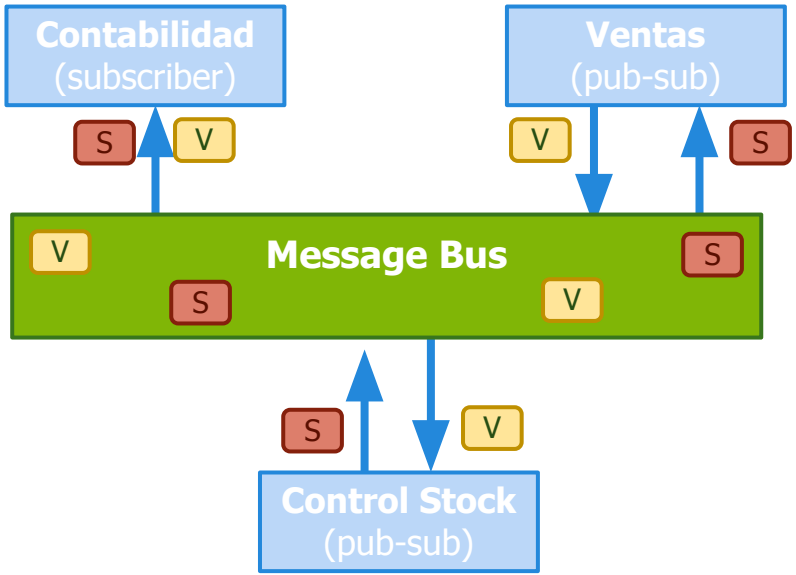
Dos posibles arquitecturas

- **Basada en tópicos:** publicación y subscripción indicando el tipo de evento, tópico o tag.
- **Basada en Canales:** publicaciones y suscripciones orientadas a canales específicos.

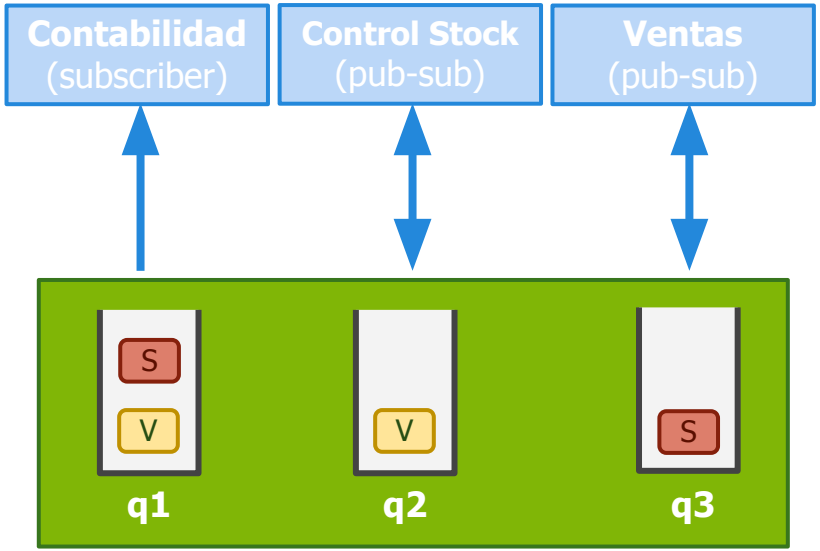


Publisher-Subscriber | Implementación con MOMs

Bus
(basado en topics)

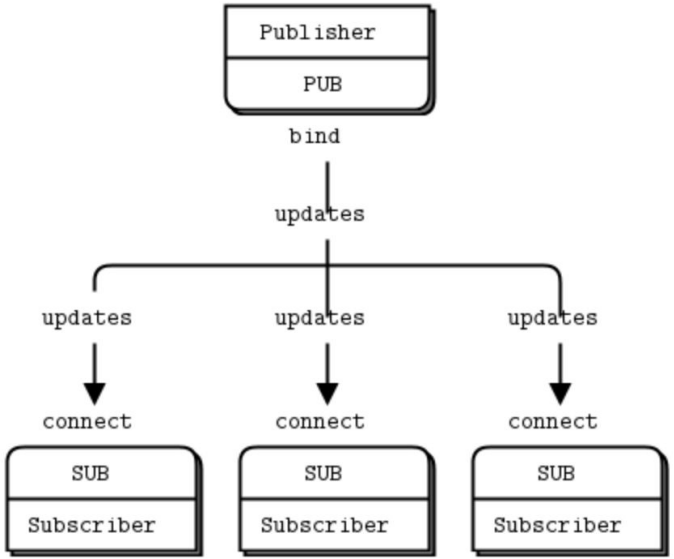


Colas
(basado en canales)



ZeroMQ | Patrones | Publisher-Subscriber

- Un ZMQ PUB socket publica mensajes
- Message pattern: `id field1 field2 ...fieldN`
- N ZMQ SUB sockets se suscriben a los Eventos que desean recibir suscribiendose al ID del evento
- Suscripción puede ser cancelada en cualquier momento
- Mensaje es enviado a todos los sockets suscriptos a un evento determinado
- Múltiples publishers? [XPUB-XSUB pattern]



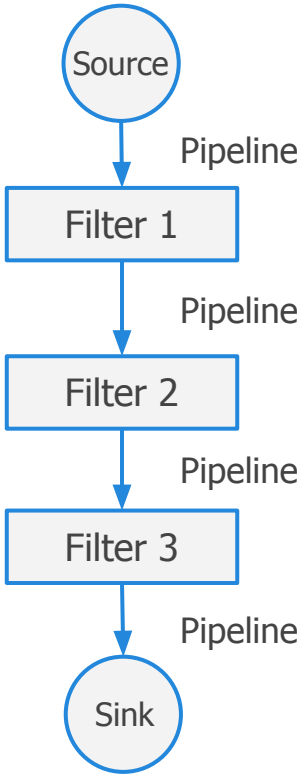
Agenda

- Request-Reply
- Publisher-Subscriber
- **Pipelines y DAGs**

Pipelines | Introducción

- En arquitecturas de software se lo conoce como *'Pipelines and Filters'*
- Los datos de entrada forman un flujo donde distintos *filters* (o *processors*) se conectan entre sí para procesarlos de manera secuencial
- Inspirado en patrones de procesamiento de señales, es muy utilizado en entornos Unix:

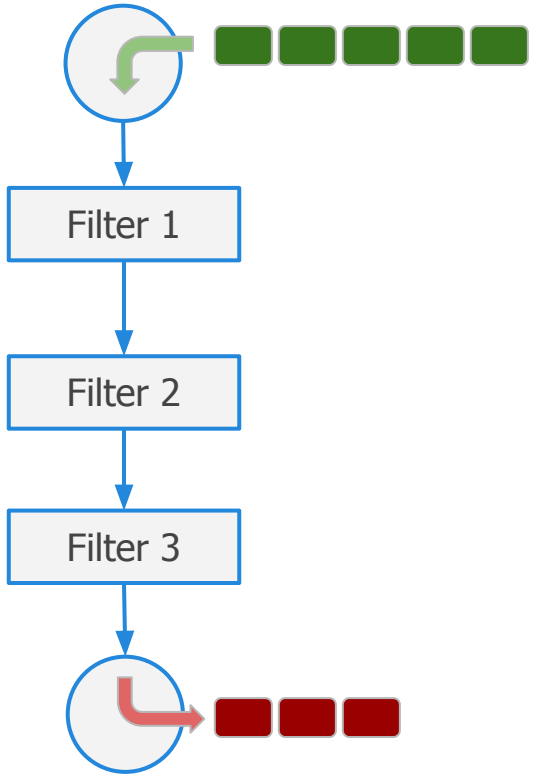
```
cat in | grep pattern | sort | uniq > out
```



Pipelines | Modelo de Procesamiento

Admite dos modelos de procesamiento:

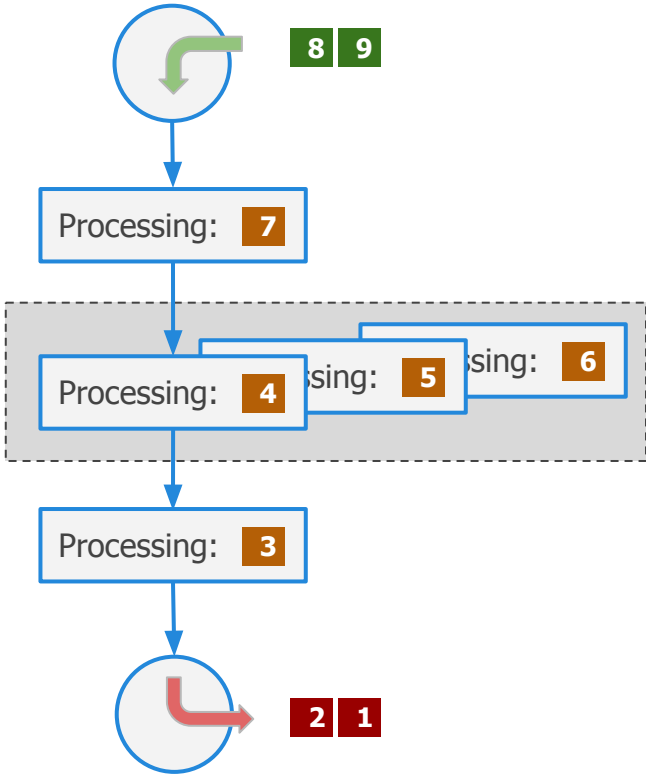
- ***Worker* por *Filter:*** se asigna una unidad de procesamiento a cada etapa del *pipeline*. Los items son recibidos por el *worker*, procesados y enviados a la próxima etapa.
- ***Worker* por Item:** se asigna una unidad de procesamiento a cada item. Un worker toma al item ingresado y lo acompaña hasta el final del pipeline, aplicándole los *filters* paso a paso.



Pipelines | Etapas secuenciales y paralelas

Cada uno de los *processors* funciona como una etapa, pudiendo ser del tipo:

- **Paralela:** cada item a procesar es independiente de anteriores y posteriores por lo que admite paralelismo.
- **Secuencial:** no puede procesar más de un item a la vez. Ya procesados, los puede retornar:
  - Ordenados
  - Desordenados

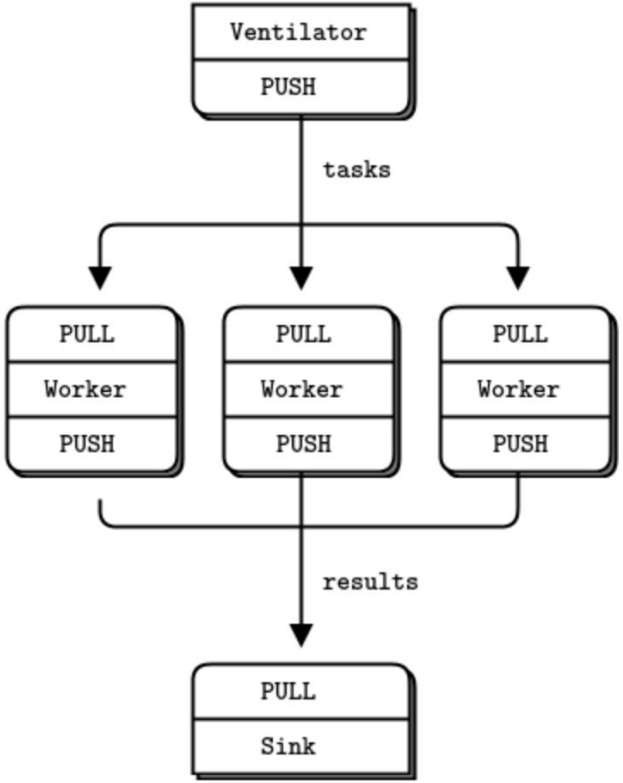


Pipelines | Ventajas

- **Algoritmos *Online:***
  - Permite iniciar el procesamiento antes de que estén disponibles todos los datos
- **Información Infinita:**
  - Permite trabajar con flujos ilimitados de información con cantidades constantes de memoria.
  - El procesamiento está encadenado, con un buffer mínimo para la configuración del pipeline.

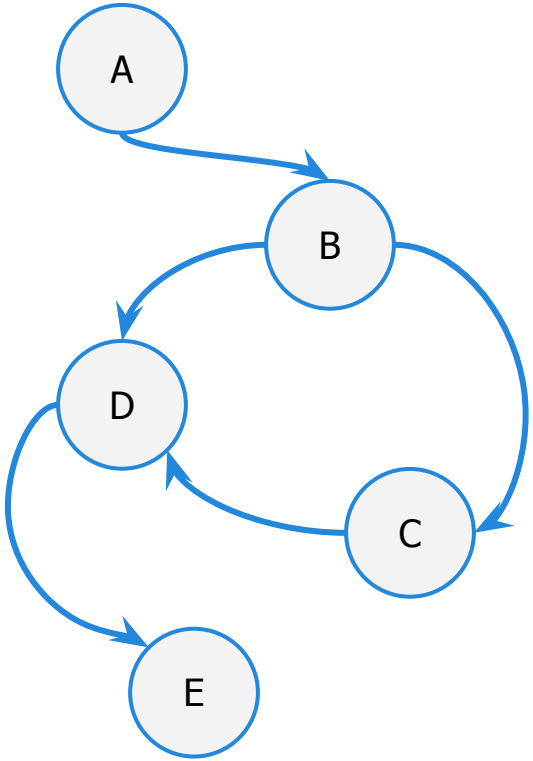
ZeroMQ | Patrones | Pipeline (Push - Pull)

- Patrón Productor-Consumidor
- Chaining de Productores-Consumidores da como resultado un pipeline
- Mensajes son consumidos de forma Equitativa **(fairness)**
  - Qué lógica utiliza para decidir esto?
- Combinaciones
  - 1 PUSH -> N PULL
  - N PUSH -> 1 PULL



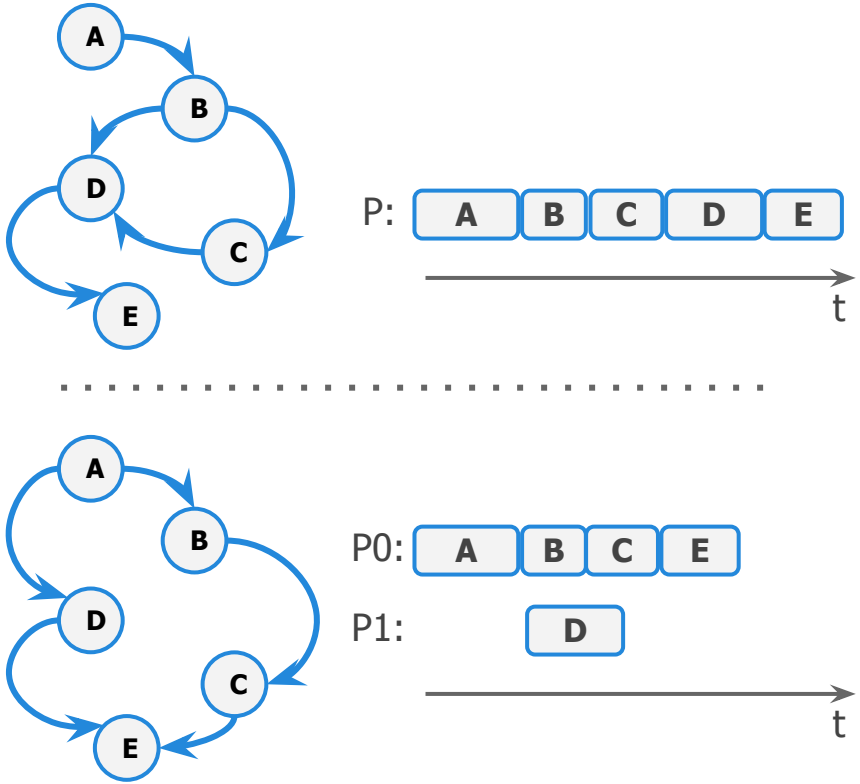
Direct Acyclic Graphs (DAGs) | Introducción

- Se modelan las instrucciones mediante un grafo de flujo de datos
- Los nodos indican tareas y las aristas el flujo de información
- Acíclicos: para todo nodo, no hay un camino que inicie y termine en él.
- Permite calcular trabajo total para cierta secuencia de tareas, camino crítico,



DAGs | Ventajas

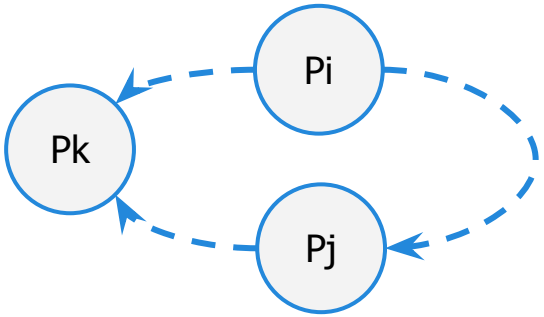
- Representación natural para *dataflows*
- La carga de procesamiento se puede paralelizar
- Admite *Lazy Loading* de las operaciones:
  - Sólo procesa nodos requeridos por dependencias



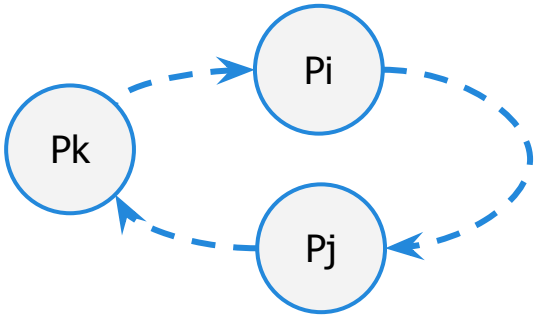
DAGs | Dependencias y *non-DAGs*

- También se pueden utilizar DAGs para modelar dependencias entre procesos
- Dependencias implican posibilidad de bloqueo frente al pedido del recurso de un proceso a otro
- Si el grafo es cíclico, existe posibilidad de *deadlock*
- Nos sirve para detectar y recuperar sistemas frente a *deadlocks*

Wait-for (DAG)



Wait-for (non DAG: Deadlock)

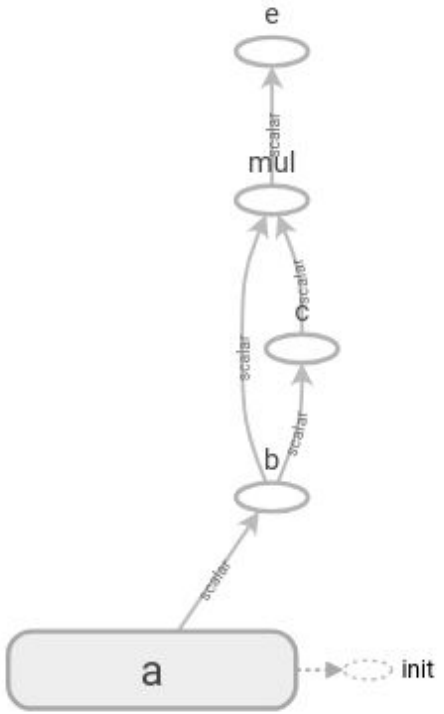


DAG | Ejemplo: Tensorflow

```
import tensorflow as tf
a = tf.get_variable('a', shape=[],
initializer =
  tf.truncated_normal_initializer(
    mean=0, stddev=1))
b = tf.negative(a, name='b')
c = tf.abs(b, name='c')
d = b * c
e = tf.abs(d, name='e')

with tf.Session() as sess:
  result = sess.run(e)

print(result)
```



Bibliografía

- G. Coulouris, J. Dollimore, t. Kindberg, G. Blair: Distributed Systems. Concepts and Design, 5th Edition, Addison Wesley, 2012.
  - Capítulo 5.2: Request-Reply Protocols
  - Capítulo 6.3: Publisher-Subscribe systems
- M. Van Steen, A. Tanenbaum: Distributed Systems. 3rd Edition. Pearson Education, 2017.
  - Capítulo 2.1: Architectural Styles, Publish-Subscribe architectures
- McCool M., Robison A. D., Reinders J., Structured Parallel Programming Patterns for Efficient Computation, 2012, Elsevier-Morgan Kaufmann.
  - Capítulo 9: Pipelines