

Computação na nuvem

ISEL – LEIRT / LEIC / LEIM

Comunicação por eventos (mensagens)

Serviço Google *Pub/Sub*

José Simão jsimao@cc.isel.ipl.pt ; jose.simao@isel.pt

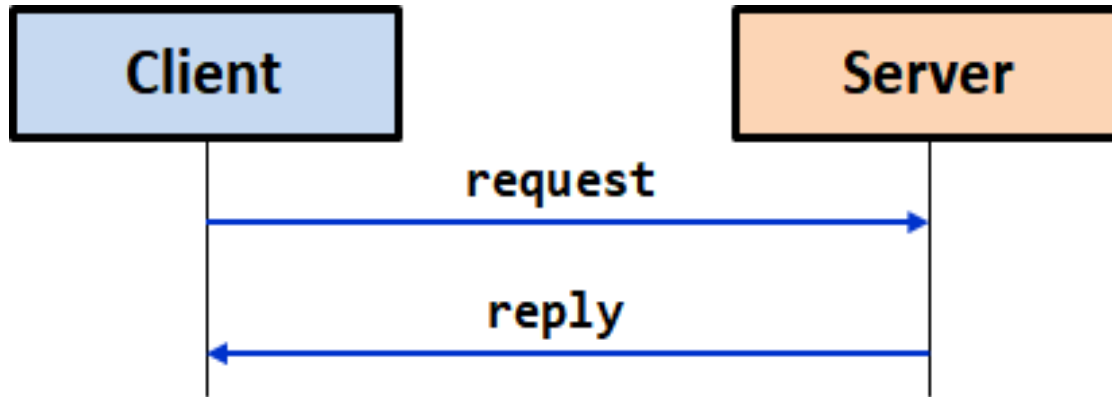
Luís Assunção lass@isel.ipl.pt ; luis.assuncao@isel.pt

Sumário

- Comunicação por eventos (mensagens)
 - *Message Oriented Middleware* (MOM);
 - Padrão *Publish/Subscribe*
- Serviço Google *Pub/Sub*
 - Conceitos e modelo de dados
 - Exemplos com a API Java do serviço Pub/Sub

Modelo *Client/Server*

- O modelo *Client/Server* (*request/reply*) é o mais usado nas arquiteturas de sistemas distribuídos



Desvantagens:

- Interação limitada a dois participantes
- Cada um tem de conhecer os endereços do outro
- Os dois participantes têm de estar presentes no mesmo tempo
- A comunicação é unicamente *pull-based* e inerentemente síncrona

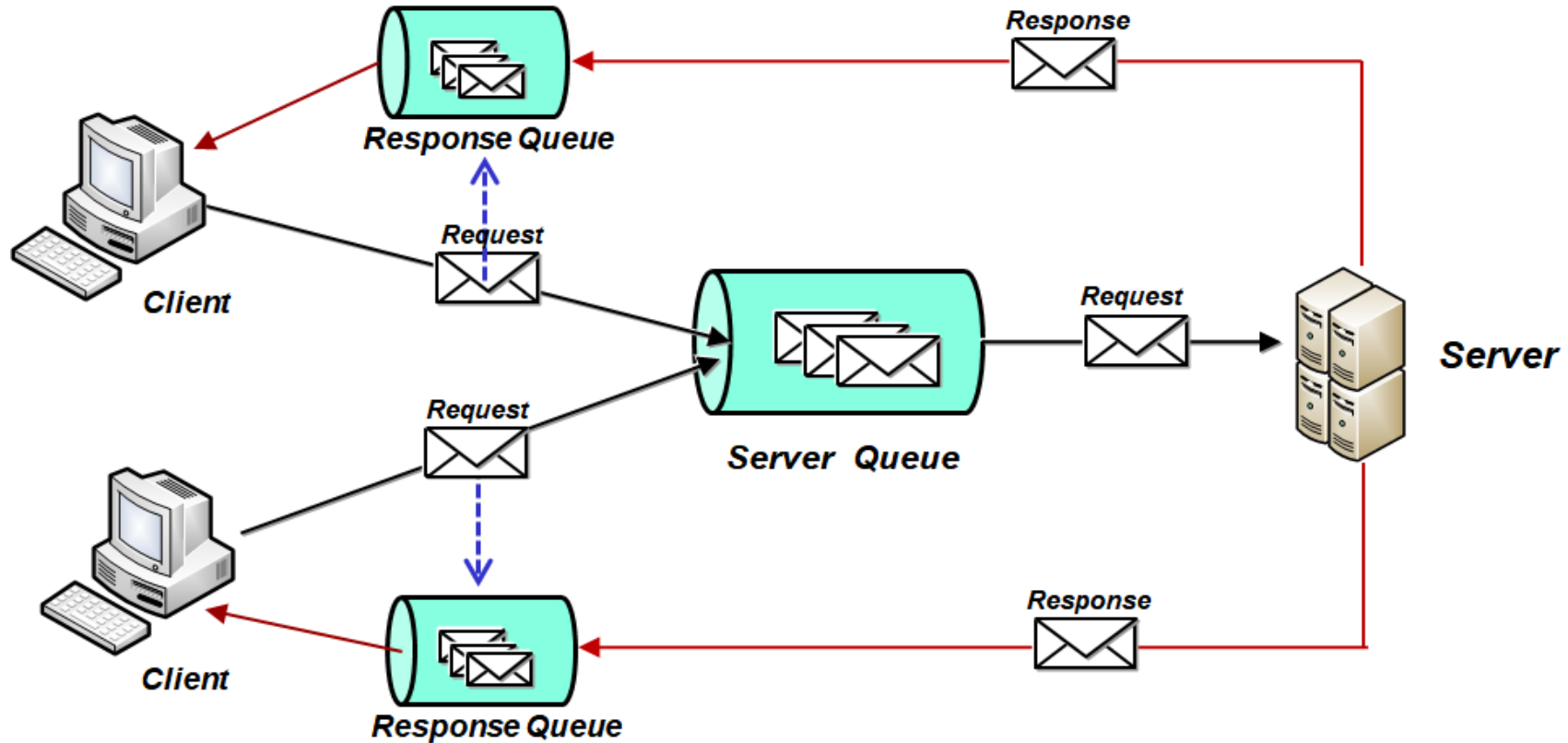
Outros Modelos

- Para resolver algumas das desvantagens da comunicação direta *Client/Server*, foram desenvolvidos modelos de comunicação por eventos/mensagens:
 - RPC sobre canais *full-duplex* com suporte de *streaming* e comunicação assíncrona (caso gRPC);
 - Filas de mensagens (*Message queues*)
 - *Publish/Subscribe* de eventos/mensagens

Comunicação por eventos (mensagens)

- Desacoplamento (*loosely coupled*) entre produtores e consumidores de eventos ou mensagens num sistema distribuído
- Diferentes ritmos de produção/consumo de eventos (mensagens). Contrariamente ao modelo *Request/Reply* existe assincronismo entre enviar uma mensagem com a sua receção. Pressupõe a existência de armazenamento das mensagens (tipicamente filas) classificadas por tópicos
- Os produtores enviam mensagens para filas ou tópicos
- Um único ou múltiplos consumidores podem receber as mensagens de uma fila ou tópico
- Como desvantagem, esta comunicação requer a existência de *middleware* intermediário designado por *Broker* ou *Mediator*

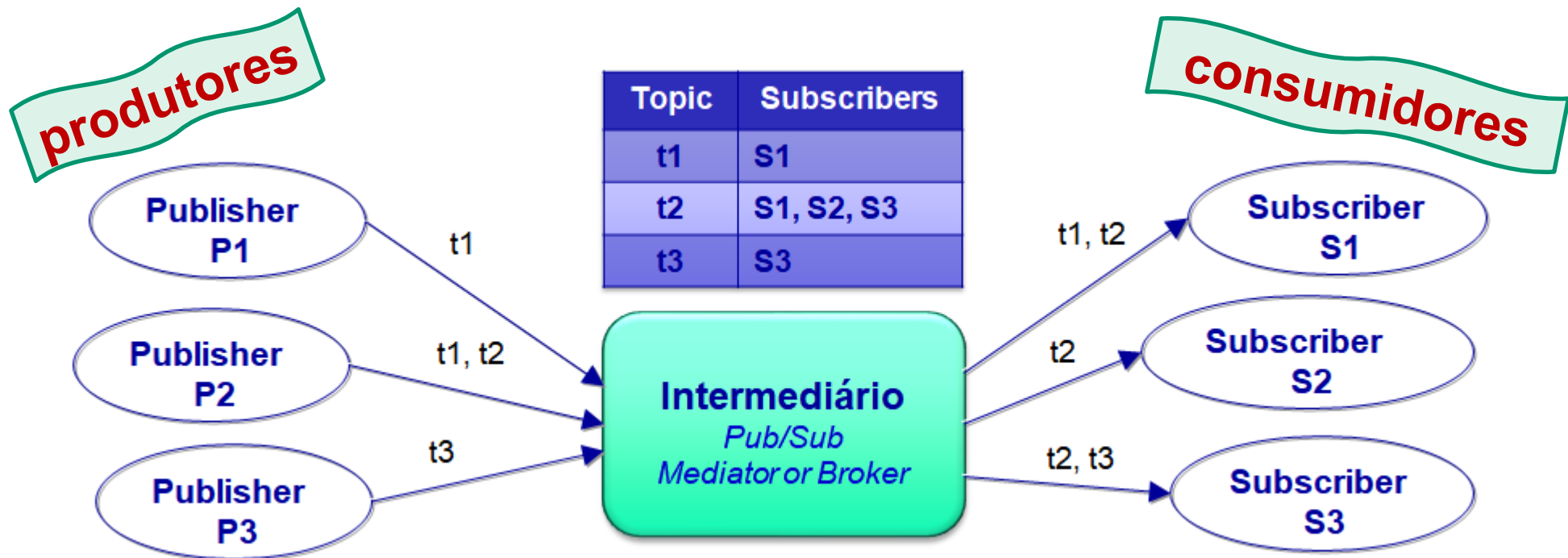
Exemplo de Filas



- Cada cliente envia na mensagem de *request* o nome da sua fila para respostas, podendo desconectar-se sem perder as respostas
- Vários servidores partilham uma fila global para processar pedidos, permitindo balanceamento de carga e tolerância a falhas

Modelo *Publish/Subscribe*

- Um intermediário mantém estado sobre um conjunto de tópicos e o conjunto de subscritores de cada tópico
- Cada subscritor pode subscrever um ou mais tópicos



Exemplos de Sistemas de Mensagens

- Java Message System (JMS);
- Microsoft Message Queue (MSMQ); IBM WebSphere MQSeries; TIBCO Rendezvous;
- *Advanced Message Queuing Protocol (AMQP)* - Especificação aberta standard (*Fedora AMQP; Apache Qpid; RabbitMQ; etc.*);
- Apache Kafka
- *ZeroMQ*
 - ❖ Instalação/Configuração complexa e nalguns casos com limitação de escala
- Cloud (sistemas de mensagens em larga escala):
 - Cloud Amazon SQS (*Simple Queue Service*); *Azure Queue Storage*; *Google Pub/Sub*

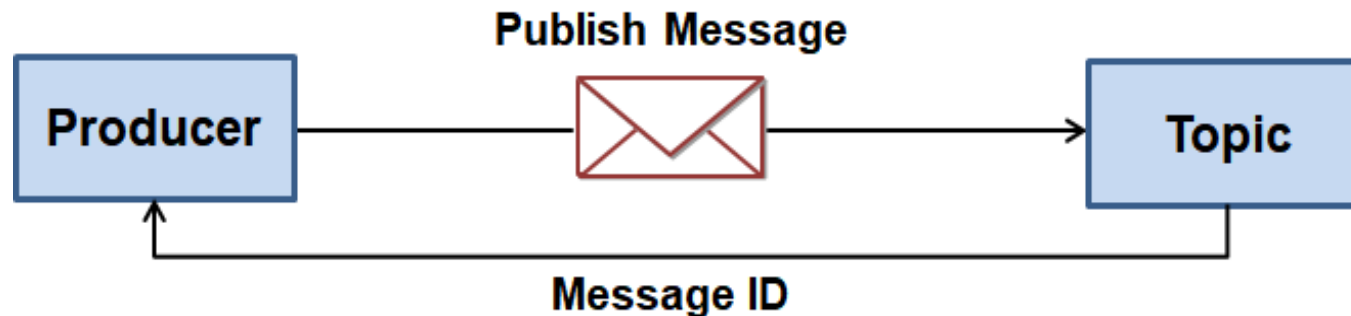
Apache Kafka

- Originalmente desenvolvido pelo LinkedIn, é um sistema distribuído *publish/subscribe* de mensagens, muito usado no processamento de eventos em *stream*.
- Como outros sistemas de mensagens o Kafka facilita a troca assíncrona de dados entre parceiros num sistema distribuído.
- Contrariamente a outros sistemas o Kafka usa uma política de retenção de mensagens (por omissão 7 dias) apagando as mensagens ao fim desse período, isto é, os consumidores são responsáveis por controlar as mensagens que foram lidas.
- O Kafka executa-se em múltiplos servidores em cluster em que cada nó é designado como *broker*, usando o serviço de coordenação distribuída *ZooKeeper*.
 - <https://kafka.apache.org/>
 - Effective Kafka, A Hands-on Guide to Building Robust and Scalable Event-Driven Applications with Code Examples in Java, Emil Koutanov, Lean Publishing, 2019-2021

Serviço Google Pub/Sub

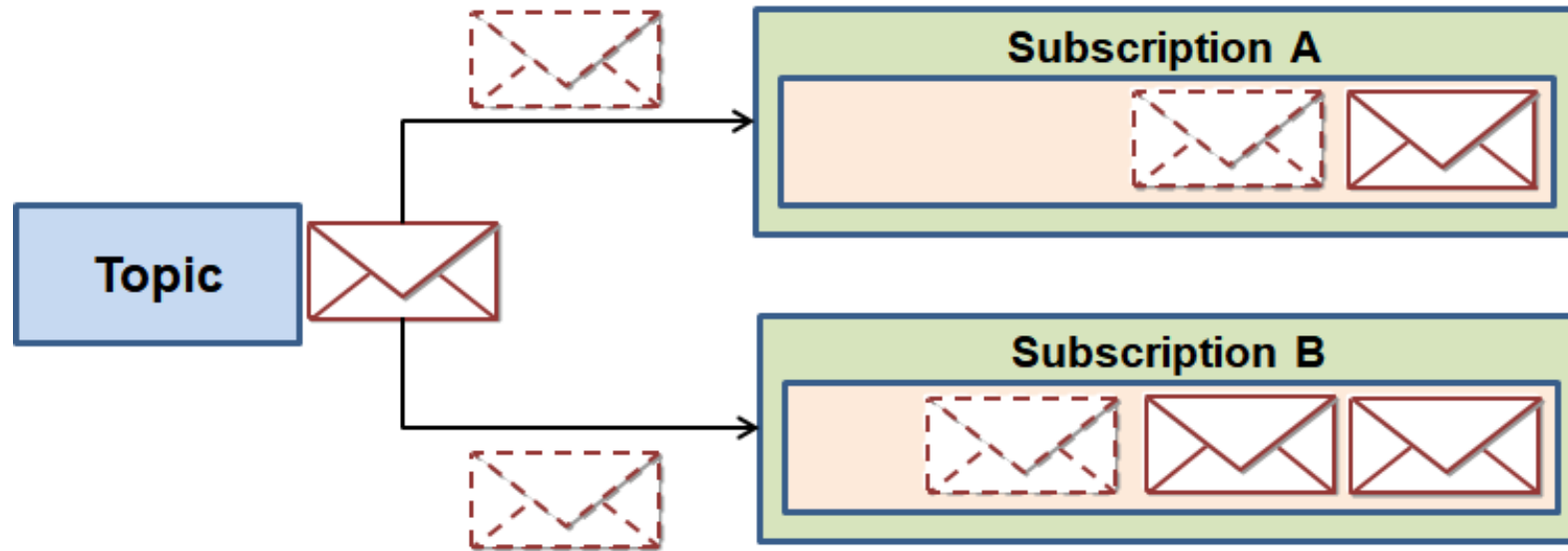
Fluxo de uma mensagem no Pub/Sub

- ❖ O fluxo de mensagens desde o produtor até ao(s) consumidor(es) é descrito em 6 passos:
 1. O produtor da mensagem escolhe um tópico (*Topic*) para onde pretende enviar a mensagem
 2. O produtor envia a mensagem (*Publish*) para um tópico específico
 3. A infraestrutura Pub/Sub recebe a mensagem, atribui-lhe um Identificador único (ID) no tópico e retorna-o para o produtor como confirmação da receção da mensagem



Fluxo de uma mensagem no Pub/Sub

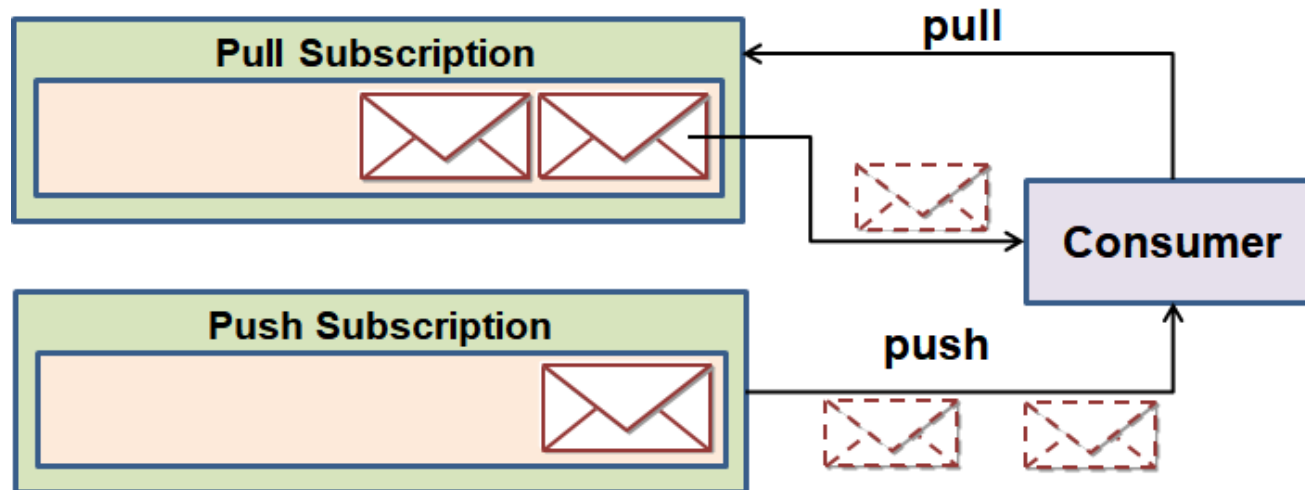
4. Um tópico pode ter uma ou mais subscrições (*Subscriptions*) associadas, pelo que as mensagens enviadas para o tópico são replicadas nas filas de cada *Subscription*.



Do ponto de vista do produtor (*publisher*) a mensagem está nas filas de todos os potenciais consumidores

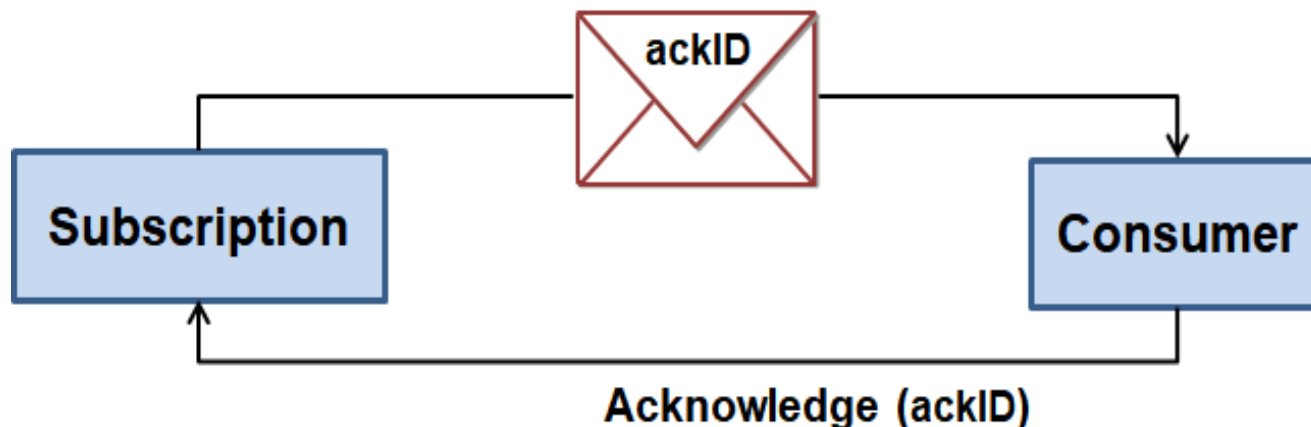
Fluxo de uma mensagem no Pub/Sub

5. As mensagens de uma subscrição podem ser entregues aos consumidores (*subscribers*) de duas formas dependentes da configuração da *Subscription*:
- **Pull**: As mensagens são retidas nas filas até os consumidores *subscribers* as irem retirar (*pull*);
 - **Push**: A subscrição ativamente envia (*push*) as mensagens para um *endpoint* pré-definido na subscrição e associado ao consumidor *subscriber*.



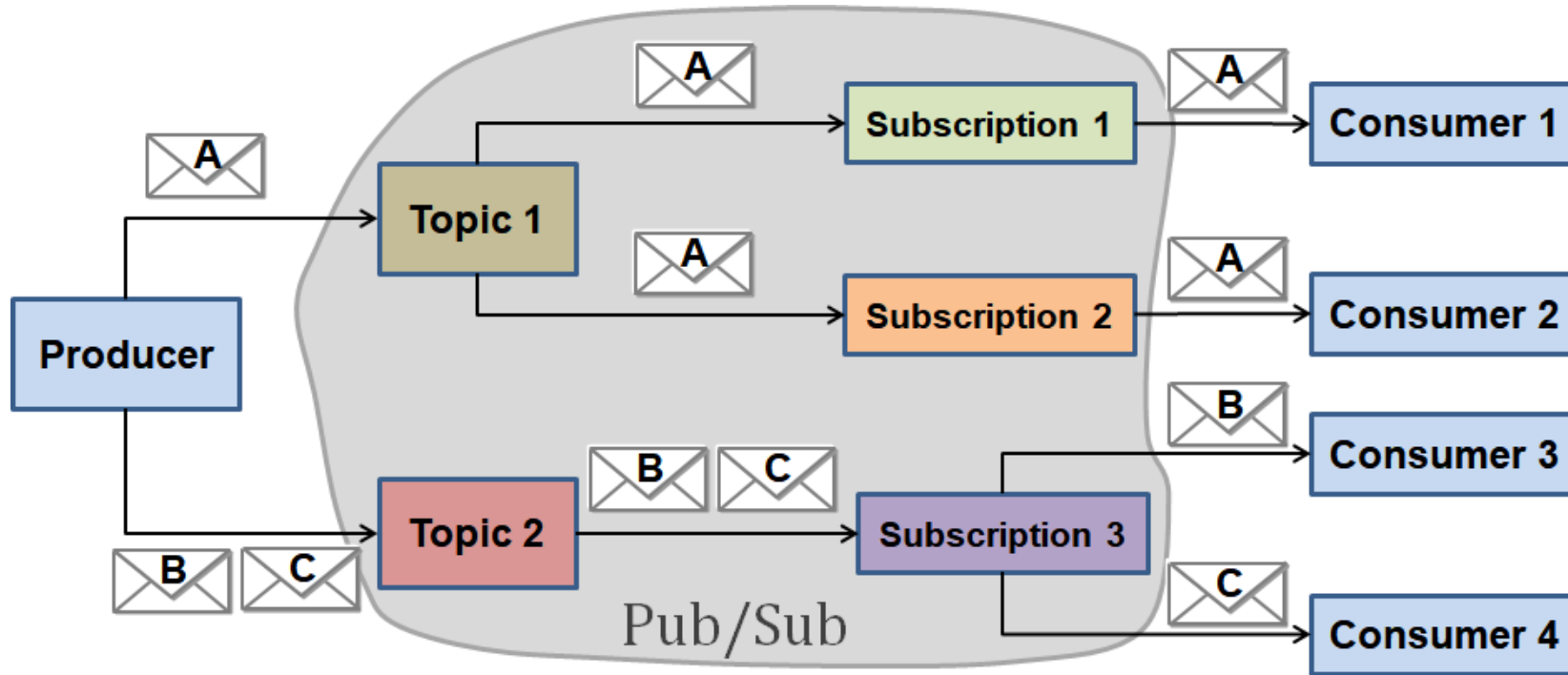
Fluxo de uma mensagem no Pub/Sub

6. Independentemente da forma (*pull* ou *push*) como a mensagem chegou ao consumidor é necessário enviar para a subscrição um **Acknowledge** da receção e processamento da mensagem.
- O *acknowledge* tem de ser dado num intervalo de tempo máximo definido na *Subscription*
 - Se o *acknowledge* não chegar no intervalo definido, a mensagem é reposta na fila como se ainda não tivesse sido entregue. Assim, evita-se perdas de mensagens, permitindo a recuperação de eventuais falhas do *Consumer* antes de completar o seu processamento



Múltiplas Subscrições e Consumidores

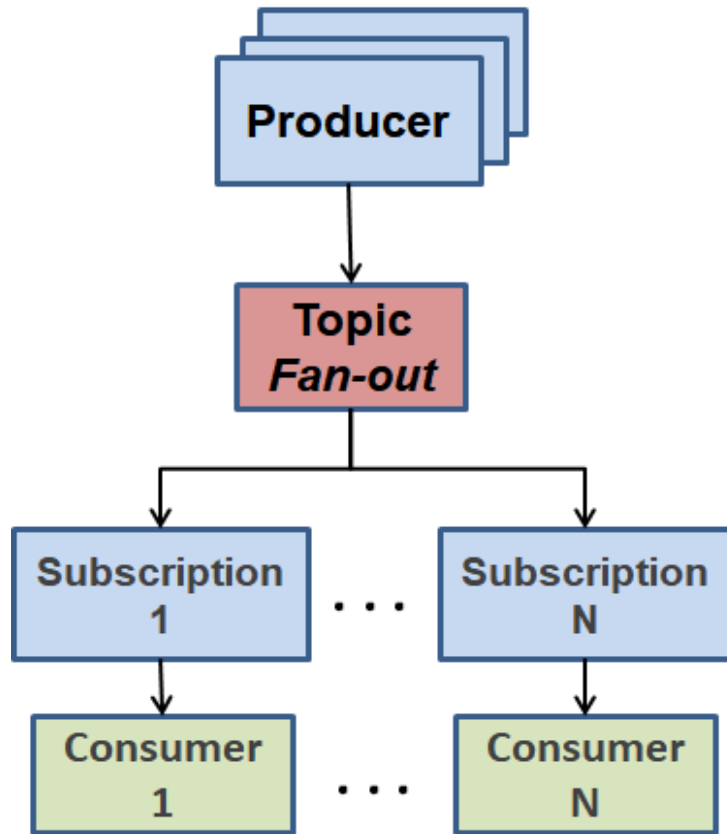
No *Topic 1*, cada *Subscription*,
recebe uma cópia da mensagem A



Na *Subscription 3*, cada mensagem é
consumida por um único *Consumer*

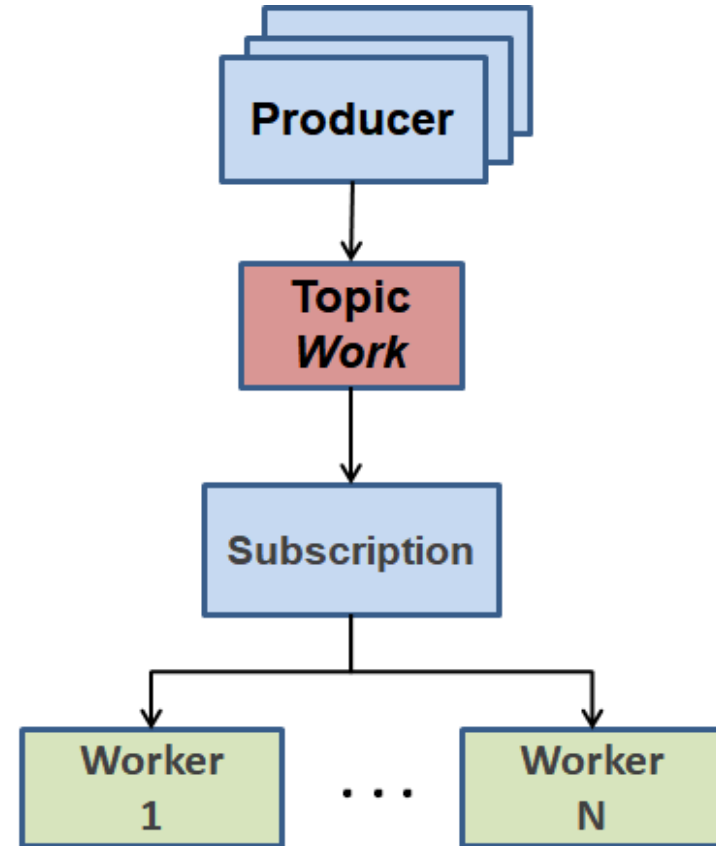
Padrões essenciais

Fan-out pattern



Difusão de informação por múltiplos consumidores

Work-queue pattern



Distribuição de trabalho por múltiplos *workers*

Conceitos: *Topic*

- Representam categorias de informação para organizar ou segmentar as mensagens (ex: departamento de uma empresa; tema de interesse de vários consumidores, etc.);
- Um tópico é um recurso do *Google Cloud Platform* definido com um nome

Create a topic




A topic forwards messages from publishers to subscribers.

Topic ID *

radares



Topic name: projects/cn2122-jsla-geral/topics/radares

- ☐ Add a default subscription 
- ☐ Use a schema 
- ☐ Set message retention duration (not free) 
- ☐ Use a customer-managed encryption key (CMEK)

CANCEL

CREATE TOPIC

Conceitos: *Schema* das mensagens

Create Schema

A schema is a format that messages must follow. You can create schemas as standalone resources, associate schemas with Pub/Sub topics, and use them to validate the structure of published messages. [Learn more](#)

Schema ID *
msgStructure

Schema name: projects/cn2122-jsla-geral/schemas/msgStructure

Schema type

☐ Avro
Avro schemas are defined with JSON.

☒ Protocol Buffer
Protocol buffer is a method of serializing structured data

Schema definition

Press Alt+F1 for Accessibility Options.

```
1 syntax = "proto2";
2
3 message ProtocolBuffer {
4   int32 msgID = 1;
5   string msgField1 = 2;
6   string msgField12 = 3;
7 }
8
```

Create Schema

A schema is a format that messages must follow. You can create schemas as standalone resources, associate schemas with Pub/Sub topics, and use them to validate the structure of published messages. [Learn more](#)

Schema ID *
msgStructure

Schema name: projects/cn2122-jsla-geral/schemas/msgStructure

Schema type

☒ Avro
Avro schemas are defined with JSON.

☐ Protocol Buffer
Protocol buffer is a method of serializing structured data

Schema definition

Press Alt+F1 for Accessibility Options.

```
1 {
2   "type" : "record",
3   "name" : "Avro",
4   "fields" : [
5     {
6       "name" : "StringField",
7       "type" : "string"
8     },
9     {
10      "name" : "IntField",
11      "type" : "int"
12    }
13   ]
14 }
15
```

No entanto, o *schema* das mensagens pode ser opaco (*ByteString*), ficando ao critério dos *producers* e *consumers* acordarem num formato específico da aplicação

Conceitos: *Message*

- Publicada num *Topic*
- Composta por um campo (*body*) de dados arbitrários (*payload*) codificados em *encoded base-64* de acordo com o *schema* se este existir
- Opcionalmente uma mensagem pode ter *metadata* através de um conjunto de atributos de texto (*Map<String, String>*)

Topic name
projects/curious-cistern-269712/topics/musica

Message body ?
my top music

Test message

Select a message encoding *
JSON

Message body
The message you want to test against the schema.

Message *
{
 "StringField" : "ola",
 "IntField" : 20
}

Avro schema

Message size should not exceed 10MB.

The message you want to publish to this topic. Either message or attribute will be required to publish.

Message attributes ?

Key	Value
Top#1	Comfortably Numb
Top#2	Time

PUBLISH CANCEL

Campos de uma mensagem

Fields	https://cloud.google.com/pubsub/docs/reference/rest/v1/PubsubMessage
data	<p>string (bytes format)</p> <p>The message data field. If this field is empty, the message must contain at least one attribute.</p> <p>A base64-encoded string.</p>
attributes	<p>map (key: string, value: string)</p> <p>Attributes for this message. If this field is empty, the message must contain non-empty data. This can be used to filter messages on the subscription.</p> <p>An object containing a list of "key": value pairs. Example: { "name": "wrench", "mass": "1.3kg", "count": "3" }.</p>
messageId	<p>string</p> <p>ID of this message, assigned by the server when the message is published. Guaranteed to be unique within the topic. This value may be read by a subscriber that receives a PubsubMessage via a subscriptions.pull call or a push delivery. It must not be populated by the publisher in a topics.publish call.</p>
publishTime	<p>string (Timestamp format)</p> <p>The time at which the message was published, populated by the server when it receives the topics.publish call. It must not be populated by the publisher in a topics.publish call.</p> <p>A timestamp in RFC3339 UTC "Zulu" format, with nanosecond resolution and up to nine fractional digits. Examples: "2014-10-02T15:01:23Z" and "2014-10-02T15:01:23.045123456Z".</p>
orderingKey	<p>string</p> <p>If non-empty, identifies related messages for which publish order should be respected. If a Subscription has enableMessageOrdering set to true, messages published with the same non-empty orderingKey value will be delivered to subscribers in the order in which they are received by the Pub/Sub system. All PubsubMessages published in a given PublishRequest must specify the same orderingKey value.</p>

Conceitos: *Subscription*

- Representa uma intenção de receber mensagens de um determinado tópico
- Uma *subscription* tem associada uma fila de retenção de mensagens até as mesmas serem consumidas.
- Uma *subscription* pode ter o modo *Pull* ou o modo *Push*
- Um tópico pode ter várias *subscriptions*. Uma mensagem enviada para um tópico é replicada por cada *subscription*.
- Um consumidor de mensagens obtém a mensagem de uma *subscription* sem afetar as restantes *subscriptions*
- Uma *subscription* pode ter múltiplos consumidores. Quando um consumidor consome uma mensagem, a mesma deixa de estar disponível para outro consumidor, isto é, cada consumidor consome diferentes mensagens da *subscription*

Criação de *Subscription* na consola GCP

Subscription ID *
topicApp-sub3

Subscription name: projects/pjbase-cn2021/subscriptions/topicApp-sub3

Select a Cloud Pub/Sub topic *
projects/pjbase-cn2021/topics/topicApp

Delivery type

☒ Pull
☐ Push

Message retention duration

Duration is from 10 minutes to 7 days

Days: 7
Hours: 0
Minutes: 0

☐ Retain acknowledged messages

When enabled, acknowledged messages are retained for the message retention duration specified above. This increases message storage fees. [Learn more](#)

Expiration period

☒ Expire after this many days of inactivity (up to 365)

31 Days

A subscription is inactive if there is no subscriber activity such as open connections, active pulls, or successful pushes.

☐ Never expire

The subscription will never expire no matter the activity.

Acknowledgement deadline

10 Seconds

Deadline time is from 10 seconds to 600 seconds

Mínimo: 10 seg..
Máximo: 600 seg. (10 min)

Ver as mensagens numa *Subscription*

Messages

To view messages published to this topic, select or create (recommended for testing) a **Pull** subscription.

Select a Cloud Pub/Sub subscription *
projects/curious-cistern-269712/subscriptions/musica-subscription

i Click **Pull** to view messages and temporarily delay message delivery to other subscribers. Select **Enable ACK messages** and then click **ACK** next to the message to permanently prevent message delivery to other subscribers. Click **Pull** again to retrieve more messages from the backlog. Use this option cautiously in production environments. If the acknowledgement deadline (10 seconds) expires, the message will be sent again if no other subscribers of this subscription acknowledge it.

PULL

☒ Enable ack messages

☰ Filter table

Publish time	Attribute keys	Message body	Ack ↑
May 10, 2020, 12:16:42 PM	—	outra mensagem de muisca	ACK
May 10, 2020, 12:13:16 PM	—	mais uma mensagem de musica	Deadline exceeded

Duas mensagens na *subscription*:

- Uma espera *acknowledge*
- Outra com o tempo de *acknowledge* já expirado

Conceitos: *Acknowledge* das mensagens

- Se uma mensagem tiver *acknowledge* positivo a mensagem é retirada da fila associada à *subscription*
- Se uma mensagem tiver *acknowledge* negativo, ou não receber *acknowledge* dentro do período de *deadline*, a mensagem é retida na fila e será reenviada novamente de acordo com a política de *Retry* definida na *subscription*

Retry policy

Retry policy will be triggered on NACKs or acknowledgement deadline exceeded events for a given message. [Learn more](#)

- ☐ Retry immediately
- ☒ Retry after exponential backoff delay

Minimum backoff (seconds) *

10

Maximum backoff (seconds) *

600

Duration is between 0 and 600 seconds

Outras características das subscriptions

Subscription filter

If a filter syntax is provided, subscribers will only receive messages that match the filter.

[Learn more](#)

```
Example: attributes:k OR (attributes.k1 = "v" AND NOT  
hasPrefix(attributes.k2, "v"))
```

Max 256 characters. Filters cannot be changed or removed once applied.

Exactly once delivery PREVIEW

☐ Enable exactly once delivery

When enabled, messages sent to the subscription are guaranteed not to be resent before the message's acknowledgement deadline expires. Acknowledged messages will not be resent to the subscription. Default acknowledgement deadline will be increased to the recommended minimum of 60 seconds. [Learn more](#)

Message ordering

☐ Order messages with an ordering key

When enabled, messages tagged with the same ordering key will be received in the order they are published. This option cannot be changed later.

Dead lettering

☐ Enable dead lettering

Subscriptions may configure a maximum number of delivery attempts. When a message cannot be delivered, it is republished to the specified dead letter topic.

As mensagens que não obedecem ao filtro são descartadas na *subscription*

Filtering syntax

To filter messages, write an *expression* that operates on attributes. You can write an expression that matches the key or value of the attributes. The `attributes` identifier selects the attributes in the message.

For example, the filters in the following table select the `name` attribute:

Filter	Description
<code>attributes:name</code>	Messages with the name attribute
<code>NOT attributes:name</code>	Messages without the name attribute
<code>attributes.name = "com"</code>	Messages with the name attribute and the value of com
<code>attributes.name != "com"</code>	Messages without the name attribute and the value of com
<code>hasPrefix(attributes.name, "co")</code>	Messages with the name attribute and a value that starts with co
<code>NOT hasPrefix(attributes.name, "co")</code>	Messages without the name attribute and a value that starts with co

Garantia de que não há reenvio duplicado das mensagens

Permite marcar as mensagens com uma *key*, garantindo a ordem das mensagens com que foram publicadas

Permite definir um número máximo de tentativas para reenviar as mensagens e um tópico para enviar as mensagens não entregues

Push Subscription

- No caso de uma *Push Subscription* a entrega da mensagem é realizada por um pedido HTTP POST para um único *endpoint* (URL) indicado na configuração da *subscription*, isto é, uma *subscription Push* só tem um consumidor

POST <https://www.example.com/my-push-endpoint>

```
{  
  "message": {  
    "attributes": {  
      "key": "value"  
    },  
    "data": "SGVsbG8gQ2xvdWQgUHVlL1N1YiEgSGVyZSBpcyBteSBtZXNzYWdlIQ==",  
    "messageId": "136969346945"  
  },  
  "subscription": "projects/myproject/subscriptions/mysubscription"  
}
```

<https://cloud.google.com/pubsub/docs/push>

The push endpoint must be a HTTPS server, presenting a valid SSL certificate

Conclusões

❖ *Vantagens*

- Desacoplamento entre produtores e consumidores de mensagens
- Diferentes ritmos de produção/consumo de eventos/mensagens
- Um único ou múltiplos consumidores podem receber as mensagens do tópico

❖ *Desvantagens*

Reliability: Não existe uma garantia forte para o *Publisher* de que todos os *subscribers* receberam a mensagem. No caso Google Pub/Sub há garantia para o *middleware* de que o *subscriber* recebe a mensagem (*acknowledge*)

Potencial bottleneck: Perante o aumento de *subscribers* e *publishers* o *Broker* pode atingir pontos de sobrecarga. Este problema é resolvido com técnicas de *load balancing*

Security: Encriptação difícil quando o *Broker* tem de interpretar o contexto das mensagens para efeitos de encaminhamento ou filtragem. O *Broker* pode amplificar os ataques de *denial of service* ao enviar as mensagens para todos os *subscribers*

Google Pub/Sub JAVA API

<https://googleapis.dev/java/google-cloud-pubsub/latest/index.html>

```
<dependencies>  
  <dependency>  
    <groupId>com.google.cloud</groupId>  
    <artifactId>google-cloud-pubsub</artifactId>  
    <version>1.116.4</version>  
  </dependency>  
</dependencies>
```

Autenticação

- Para executar aplicações Java de acesso ao *PubSub* deve definir-se a seguinte variável de ambiente:

`GOOGLE_APPLICATION_CREDENTIALS="pathname do Account service KEY.json"`

// A API depende do valor do identificador do projeto

// Nos exemplos de código a seguir existe uma constante

// PROJECT_ID que assume que tem esse identificador

Resumo da API

TopicAdminClient: Criar e listar tópicos e subscrições do tópico

TopicName: Nome do Tópico num projeto *TopicName.ofProjectTopicName()*

Topic: Tópico

Publisher: produtor de mensagens, associado a um tópico específico

PubsubMessage: Mensagem Pub/Sub

SubscriptionAdminClient: criar e listar *Subscriptions*

Subscription: subscrição *pull* ou *push* sobre um tópico

Subscriber: consumidor de mensagens associado a uma *subscription*

```
interface MessageReceiver {  
    void receiveMessage(PubsubMessage msg, AckReplyConsumer msgAck);  
}  
  
interface AckReplyConsumer {  
    void ack();    void nack();  
}
```

Criar e Listar Tópicos

```
TopicAdminClient topicAdmin = TopicAdminClient.create();
```

```
// CRIAR
```

```
TopicName tName=TopicName.ofProjectTopicName(PROJECT_ID, topicName);
```

```
Topic topic=topicAdmin.createTopic(tName);
```

```
// LISTAR
```

```
TopicAdminClient.ListTopicsPagedResponse res =
```

```
    topicAdmin.listTopics(ProjectName.of(PROJECT_ID));
```

```
for (Topic top : res.iterateAll()) {
```

```
    System.out.println("TopicName=" + top.getName());
```

```
}
```

```
topicAdmin.close();
```

nome atribuído
ao tópico



Criar Subscription

```
TopicName tName=TopicName.ofProjectTopicName(PROJECT_ID, topicName);  
SubscriptionName subscriptionName =  
    SubscriptionName.of(PROJECT_ID, subsName);  
SubscriptionAdminClient subscriptionAdminClient =  
    SubscriptionAdminClient.create();  
PushConfig pconfig=PushConfig.getDefaultInstance();  
//PushConfig.newBuilder().setPushEndpoint(ConsumerURL).build();  
Subscription subscription =  
    subscriptionAdminClient.createSubscription(  
        subscriptionName, topicName, pconfig, 0);  
  
subscriptionAdminClient.close();
```

default deadline 10 seg

Publicar mensagens num tópico

```
TopicName tName=TopicName.ofProjectTopicName(PROJECT_ID, topicName);  
Publisher publisher = Publisher.newBuilder(tName).build();  
// Por cada mensagem  
ByteString msgData = ByteString.copyFromUtf8(msgTxt);  
PubsubMessage pubsubMessage = PubsubMessage.newBuilder()  
    .setData(msgData)  
    .putAttributes("key1", "value1")  
    .build();  
ApiFuture<String> future = publisher.publish(pubsubMessage);  
String msgID = future.get();  
System.out.println("Message Published with ID=" + msgID);  
// No fim de enviar as mensagens  
publisher.shutdown();
```



String com mensagem

Criar *Subscriber* (consumidor)

```
ProjectSubscriptionName subscriptionName =  
    ProjectSubscriptionName.of(PROJECT_ID, subName);  
  
ExecutorProvider executorProvider =InstantiatingExecutorProvider  
    .newBuilder()  
    .setExecutorThreadCount(1) // um só thread no handler  
    .build();  
  
Subscriber subscriber =  
    Subscriber.newBuilder(subscriptionName, new MessageReceiveHandler())  
        .setExecutorProvider(executorProvider)  
        .build();  
subscriber.startAsync().awaitRunning();
```

Para terminar a subscrição (*UnSubscriber*)
`subscriber.stopAsync();`

Handler de receção de mensagens

```
public class MessageReceiveHandler implements MessageReceiver {  
  
    public void receiveMessage(PubsubMessage msg, AckReplyConsumer ackReply)  
    {  
        System.out.println("Message (Id:" + msg.getMessageId()+  
                             " Data:"+msg.getData().toStringUtf8()+")");  
  
        // dar acknowledge só após se ter processado a mensagem  
        ackReply.ack(); // acknowledge positivo  
    }  
}
```

Exemplo de *acknowledge* negativo

➤ Classe *Handler* de receção de mensagens

```
// This example is for illustration. Implementations may directly
// process messages instead of sending them to queues.

// Fila com capacidade máxima para 100 mensagens
BlockingQueue<PubsubMessage> queue =
    new ArrayBlockingQueue<PubsubMessage>(100);

MessageReceiver receiver = new MessageReceiver() {
    public void receiveMessage(PubsubMessage msg,
                               AckReplyConsumer ackReply) {
        if (queue.offer(msg)) {
            ackReply.ack(); // acknowledge positive
        } else { // Queue is full
            ackReply.nack(); // acknowledge negative
        }
    }
}
```