Push Notification service like FCM

i have a question lets underdarstand my current implementation first , i have a client service (service-ui) and backend (push-server) so push-server is implemented in pure netty bootstrapserver with channelduplex handler this service is developed like FCM push notification service so we are also using activeMQ to hold message and acknowledgement flow like any device fist subscribe to queue and a connect request will be send to get any longpolling response means it will wait until message available to 'message-queue' and other sevice that is service-ui admin portal so we can send notification message to subscribed devices so what we do actually first i send message to push-server using okhttpClient.newCall(.).execute() and on success i send one more request to push-server to get acknowledgement from device whether message is send to device or now so means so here device send an ack request to push server and add status to 'ack-queue' this wasy acknowlement is received to service-ui means notification sender to confirm message is send successfully and this ack request is async send using separate thread and response added throw callbackhandlerListener so there may be a chance of late acknowlement recieved but in unfortunate case connection gets closed and service-ui gets restarted after making push request so i have to handle this case so i am thinking here to use longpolling ack request to push-server always keep request open to push server like if one ackrequest completed just send another request and wait for any acknowlement so on restart also immidieatly request will be send to push-server and if any ack pending so return as response please suggest what is you point of view on this

**ChatGPT said:**

Thanks for the detailed explanation — your architecture is quite clear, and you're essentially building a custom push-notification delivery + ack-tracking system similar to FCM, with the following components:

**🔧 Current Setup Summary:**

* push-server: A Netty server managing long-lived device connections and message delivery (FCM-like).
* service-ui: An admin panel to send messages and track acknowledgments (ACKs).
* ActiveMQ:
  + message-queue: Holds messages to be sent to devices.
  + ack-queue: Receives ACKs from devices via push-server.

**✅ Flow:**

1. service-ui sends push notification → push-server.
2. push-server delivers to device.
3. Device responds → push-server adds ACK to ack-queue.
4. service-ui listens for ACK (via callback or later GET).
5. **Problem**: If service-ui crashes/restarts **after** sending the push and before receiving the ACK, ACK may be missed.

**🧠 Your Proposal: Long-polling ACK listener**

You're considering having service-ui **maintain a long-polling connection to push-server** that:

* Waits for ACKs.
* Immediately re-establishes after completion (infinite poll loop).
* Guarantees receipt even after crash/restart.

**✅ Pros of Your Approach**

1. **Resilience**: Survives restart. As soon as service-ui comes back up, it starts listening again.
2. **No need for persistent state tracking in service-ui** — handled by queue+Netty.
3. **Low latency**: ACK gets pushed quickly to the client without polling overhead.

**🚧 Considerations / Challenges**

**1. Scalability**

* Long-polling per user or message could strain push-server.
* If there are many concurrent requests from service-ui (e.g., multi-tenant, high volume), long-polling could get messy.

💡 **Solution**: Have a single long-polling ACK channel per service-ui instance or logical sender group (not per message).

**2. Backlog Handling**

* On restart, if an ACK has already been added to ack-queue, but service-ui missed it, **how will it be picked up**?

💡 **Solution Options**:

* Either push-server caches last few ACKs (in memory or file) until consumed.
* Or **use durable subscriber** in ActiveMQ — service-ui subscribes to ack-queue and consumes even after restart.

⚠️ *Long-polling should complement reliable message queues, not replace them.*

**3. Connection Management in Netty**

* Make sure push-server supports long-held connections without memory leaks.
* Implement **heartbeat / timeout** mechanism to avoid zombie connections.

**4. ACK Correlation**

* How do you match which ACK belongs to which message?

💡 **Use message ID or correlation ID** in:

* Push request.
* ACK response.
* Store mapping temporarily in service-ui if needed.

**✅ Recommendation**

**✅ Best Practice Mix (Reliable + Real-time):**

1. **Durable Messaging**:
   * Make service-ui a **durable consumer** of ack-queue in ActiveMQ.
   * Guarantees no ACKs are lost even if restarted.
2. **Long Polling or WebSocket** (Optional):
   * Add long-polling to push-server for **instant delivery** of ACKs.
   * On receiving ACK from device, push immediately to connected clients (e.g., service-ui), or use SSE (Server-Sent Events).
3. **Push-server ACK buffer**:
   * Keep last N ACKs in memory per client/session to support immediate response if reconnect happens after brief downtime.
4. **Fallback Polling**:
   * If long-polling fails (e.g., network partition), fall back to periodic polling for missed ACKs.

**🔁 Sample Long Poll ACK Handler Design (in Pseudo):**

java

CopyEdit

// service-ui

while (true) {

try {

Response response = httpClient.newCall(longPollAckRequest()).execute();

List<Ack> acks = parse(response);

processAck(acks);

} catch (IOException e) {

// retry after a delay

Thread.sleep(500);

}

}

**📌 Final Thoughts**

Your long-polling idea is valid **and useful for real-time feedback**, but for **durability**, **pair it with proper message queue consumption** (like a durable ActiveMQ subscription). This way, no ACK is ever lost, even in the worst-case scenarios.

Let me know if you’d like help designing the long-polling endpoint in Netty or configuring durable subscriptions in ActiveMQ.

**You said:**

yes please go ahead

**ChatGPT said:**

Great! Let’s break this down into two parts:

**✅ Part 1: Durable ACK Handling via ActiveMQ**

We'll ensure service-ui can **consume ACKs from ack-queue** reliably, even after a restart.

**🧱 1. ActiveMQ Durable Consumer Setup (for service-ui)**

If you're using JMS (Java Message Service):

**✅ Step-by-step:**

java

CopyEdit

ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory("tcp://localhost:61616");

connectionFactory.setClientID("service-ui-ack-consumer"); // Important for durability

Connection connection = connectionFactory.createConnection();

connection.start();

Session session = connection.createSession(false, Session.AUTO\_ACKNOWLEDGE);

// Durable subscription to ack topic/queue

Topic ackTopic = session.createTopic("ack-queue");

MessageConsumer consumer = session.createDurableSubscriber(ackTopic, "service-ui-subscription");

consumer.setMessageListener(message -> {

if (message instanceof TextMessage) {

String ackJson = ((TextMessage) message).getText();

processAck(ackJson); // Your logic to handle the ack

}

});

🧠 Note: If you're using a **Queue** and not a Topic, then messages are **already persisted per consumer**, so even a simple consumer works fine without the durable subscriber setup.

**✅ Part 2: Real-time ACKs via Long Polling (from push-server to service-ui)**

This improves **real-time delivery** of acknowledgments. Let’s now implement this with **Netty + OkHttp**.

**⚙️ Push-server: Long Poll ACK Endpoint (Netty Server)**

Each service-ui client can call /ack-longpoll, and the server holds the request until ACK is available or timeout occurs.

**🧱 1. Netty Handler (simplified)**

java

CopyEdit

// Map of clientId to ChannelHandlerContext

ConcurrentHashMap<String, ChannelHandlerContext> ackWaiters = new ConcurrentHashMap<>();

public class LongPollAckHandler extends SimpleChannelInboundHandler<FullHttpRequest> {

@Override

protected void channelRead0(ChannelHandlerContext ctx, FullHttpRequest req) throws Exception {

if (req.uri().startsWith("/ack-longpoll")) {

String clientId = extractClientId(req);

ackWaiters.put(clientId, ctx);

// Don't respond yet; wait for ACK to arrive

} else {

ctx.fireChannelRead(req.retain());

}

}

public void sendAckToClient(String clientId, String ackJson) {

ChannelHandlerContext ctx = ackWaiters.remove(clientId);

if (ctx != null && ctx.channel().isActive()) {

FullHttpResponse response = new DefaultFullHttpResponse(

HttpVersion.HTTP\_1\_1,

HttpResponseStatus.OK,

Unpooled.copiedBuffer(ackJson, CharsetUtil.UTF\_8)

);

response.headers().set(HttpHeaderNames.CONTENT\_TYPE, "application/json");

ctx.writeAndFlush(response).addListener(ChannelFutureListener.CLOSE);

}

}

}

**🚀 Device ACK Flow in push-server**

When a device sends an ACK:

java

CopyEdit

public void handleDeviceAck(String clientId, String ackJson) {

// 1. Push to ActiveMQ

jmsTemplate.convertAndSend("ack-queue", ackJson);

// 2. Immediately send to waiting service-ui, if any

longPollAckHandler.sendAckToClient(clientId, ackJson);

}

**🧪 service-ui: Long Polling Client (Java OkHttp)**

java

CopyEdit

while (true) {

try {

Request request = new Request.Builder()

.url("http://push-server-host/ack-longpoll?clientId=service-ui")

.build();

Response response = client.newCall(request).execute();

if (response.isSuccessful()) {

String ackJson = response.body().string();

processAck(ackJson); // Your logic

}

} catch (IOException e) {

// Log error, retry after delay

Thread.sleep(500);

}

}

**🧠 Optimizations You May Consider**

| **Feature** | **Description** |
| --- | --- |
| **Timeout** | Keep long-poll open for max 30s. If no ACK, return empty or "keep-alive", and reconnect. |
| **Batch ACK** | Return a list of ACKs if multiple are pending. |
| **Reconnect on failure** | Retry with exponential backoff if connection fails. |
| **Client ID correlation** | Ensure each service-ui instance uses a unique ID. |
| **Failover** | If using a cluster of push-servers, load balance long-polling. |
| **WebSocket Alternative** | For high-frequency ACKs, a WebSocket channel is more efficient. |

**✅ Final Design Summary**

| **Concern** | **Mechanism** |
| --- | --- |
| ACK delivery guarantee | ActiveMQ ack-queue + durable consumer |
| Real-time ACK | Long polling from service-ui to push-server |
| Recovery after crash | Long polling reconnect + message persistence |
| Server performance | Map clientId to long-polling request, one per instance |

Let me know if you want full working code for:

* Netty server
* Long-polling demo
* ActiveMQ durable consumer with Spring

Or if you're considering upgrading to WebSocket-based real-time notification tracking — happy to help with that too.

**You said:**

but i have one doubt here as general in producer consumer with queue lets someone pushed a message then consumer/listener with read that message and through backpressor response will be return to caller but lets if consumer is failed due to connection loss so i think message will be lost and if even i send another request on restart i don't thing i will get the same message again as that previous push ack event was missed so when an other ack push message event occure for new acknowlement then that will be handled so that can be handled

**ChatGPT said:**

You're **absolutely right** to be concerned — this is a **classic delivery reliability problem** in messaging systems:

**❗ The Problem You Raised:**

Let’s rephrase it to ensure clarity:

* A device **sends an ACK** to push-server.
* push-server forwards that ACK to **ActiveMQ's ack-queue**.
* service-ui (consumer) **was down or disconnected** at that moment.
* So:
  + If ack-queue is configured with **non-durable/non-persistent** messaging, the ACK is **lost**.
  + If a **new ACK** later comes in, **it will be received**, but the **original one is never recovered**, even after restart.

**✅ Goal: Guarantee that no ACK is lost, even if service-ui is restarted or crashes.**

**✅ Solution Overview**

**✅ Step 1: Use Durable Queues in ActiveMQ**

Make sure ack-queue is **persistent** and **durable**.

* **Queue (not topic)** → ensures 1:1 delivery.
* **Persistent messages** → stored on disk, not memory.
* **Auto-ack disabled** → consumer must explicitly ack.
* **Redelivery enabled** → if consumer crashes, message remains.

**✅ Step 2: Configure the Consumer to be Reliable**

In service-ui:

java

CopyEdit

ConnectionFactory factory = new ActiveMQConnectionFactory("tcp://localhost:61616");

Connection connection = factory.createConnection();

connection.start();

Session session = connection.createSession(false, Session.CLIENT\_ACKNOWLEDGE); // manual ack

Queue ackQueue = session.createQueue("ack-queue");

MessageConsumer consumer = session.createConsumer(ackQueue);

consumer.setMessageListener(message -> {

try {

String body = ((TextMessage) message).getText();

processAck(body); // Your logic here

message.acknowledge(); // Only ACK after processing

} catch (Exception e) {

// Don't ack → message will be redelivered

log.error("ACK processing failed, will retry");

}

});

**✅ Step 3: Set Message TTL Carefully**

Ensure your broker doesn't auto-expire messages **too soon**. Set the TTL of messages high enough to allow retries.

In the producer (push-server):

java

CopyEdit

Message msg = session.createTextMessage(ackJson);

msg.setJMSExpiration(0); // or set explicit TTL if needed

producer.send(msg, DeliveryMode.PERSISTENT, priority, ttl);

**🔄 So what happens if service-ui crashes?**

1. Device sends ACK → push-server → ActiveMQ ack-queue.
2. service-ui crashes before reading it.
3. Message **stays in the queue** (because:
   * DeliveryMode is PERSISTENT
   * Consumer didn’t ACK it).
4. After restart, service-ui re-subscribes → message is delivered again.

✅ **No ACK lost. No data missed. Exactly what we want.**

**🔁 Additional Best Practices**

| **Scenario** | **Strategy** |
| --- | --- |
| Multiple service-ui instances | Use **message groups** or one queue per client |
| Device ACK delay tolerance | Add **timestamp / retry** logic |
| Message deduplication | Use **correlation ID** or message UUID in ackJson |
| Network partitions | Broker handles redelivery automatically on reconnect |
| Debugging missed ACKs | Enable ActiveMQ persistence logs and DLQ (dead letter queue) |

**🔍 TL;DR**

| **Your Concern** | **Solution** |
| --- | --- |
| Consumer down → ACK missed | Use **persistent durable queue**, CLIENT\_ACKNOWLEDGE mode |
| Consumer restarted → needs old ACK | Message is re-delivered unless ACK’d |
| Avoid reprocessing same ACK twice | Use message ID/correlation ID in ackJson |

Would you like a working minimal Spring Boot setup with this durable ActiveMQ consumer? I can share that with everything ready to run.

**You said:**

can you implement all steps again inside ChannelDuplexHandler for if header is ack use flux.create(emitter) and messageListenerAdapter listenerand set delegate,default listener method, emit use SimpleMessageListenerContainer ..Queue (not topic) → ensures 1:1 delivery. Persistent messages → stored on disk, not memory. Auto-ack disabled → consumer must explicitly ack. Redelivery enabled → if consumer crashes, message remains. , Flux backpressor to wait for message and response back if message is received

**ChatGPT said:**

Absolutely! Let's walk through the **complete solution**, all implemented **inside a Netty ChannelDuplexHandler**, using:

* Flux.create(emitter) for backpressure-aware reactive handling.
* SimpleMessageListenerContainer (Spring JMS) to consume ack-queue.
* Durable, persistent ActiveMQ message configuration.
* Manual ACK handling (redelivery if consumer fails).
* Matching ack header in request to enable long-polling behavior.

**✅ Assumptions**

* You’re using **Spring Boot** (or Spring Core) with spring-jms and reactor-core.
* You want:
  + If header.ack == true → begin long-polling ACK handling via Flux.
  + One-to-one durable queue (not topic).
  + Messages persisted.
  + Manual ack and redelivery if crash.

**✅ Maven Dependencies (Spring Boot style)**

xml

CopyEdit

<dependencies>

<!-- JMS + ActiveMQ -->

<dependency>

<groupId>org.apache.activemq</groupId>

<artifactId>activemq-broker</artifactId>

<version>5.17.4</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jms</artifactId>

</dependency>

<!-- Reactor Core -->

<dependency>

<groupId>io.projectreactor</groupId>

<artifactId>reactor-core</artifactId>

</dependency>

</dependencies>

**✅ Step-by-Step Code**

**🔧 1. Create the Flux-backed Queue Listener**

java

CopyEdit

public class AckFluxEmitter {

private final Flux<String> ackFlux;

private FluxSink<String> sink;

public AckFluxEmitter(ConnectionFactory connectionFactory, String queueName) {

this.ackFlux = Flux.create(emitter -> {

this.sink = emitter;

SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();

container.setConnectionFactory(connectionFactory);

container.setQueueNames(queueName);

container.setSessionAcknowledgeMode(Session.CLIENT\_ACKNOWLEDGE);

container.setMessageListener(new MessageListenerAdapter(new Object() {

@SuppressWarnings("unused")

public void handleMessage(String ackMessage, Session session, Message rawMsg) {

try {

sink.next(ackMessage); // backpressured emit

rawMsg.acknowledge();

} catch (Exception e) {

// no ack → redelivery

sink.error(e);

}

}

}) {{

setDefaultListenerMethod("handleMessage");

}});

container.start();

}, FluxSink.OverflowStrategy.BUFFER); // BUFFER ensures backpressure safety

}

public Flux<String> getAckFlux() {

return ackFlux;

}

}

**🔧 2. Use in ChannelDuplexHandler (push-server)**

java

CopyEdit

public class AckChannelHandler extends ChannelDuplexHandler {

private final AckFluxEmitter ackFluxEmitter;

public AckChannelHandler(ConnectionFactory connectionFactory) {

this.ackFluxEmitter = new AckFluxEmitter(connectionFactory, "ack-queue");

}

@Override

public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {

if (msg instanceof FullHttpRequest request) {

HttpHeaders headers = request.headers();

if ("true".equalsIgnoreCase(headers.get("ack"))) {

handleAckLongPoll(ctx);

} else {

ctx.fireChannelRead(msg); // normal flow

}

} else {

super.channelRead(ctx, msg);

}

}

private void handleAckLongPoll(ChannelHandlerContext ctx) {

ackFluxEmitter.getAckFlux()

.next() // backpressured, returns Mono<String> — completes on first ack

.timeout(Duration.ofSeconds(30)) // timeout if no ACK comes

.subscribe(

ack -> {

FullHttpResponse response = new DefaultFullHttpResponse(

HttpVersion.HTTP\_1\_1,

HttpResponseStatus.OK,

Unpooled.copiedBuffer(ack, StandardCharsets.UTF\_8)

);

response.headers().set(HttpHeaderNames.CONTENT\_TYPE, "application/json");

ctx.writeAndFlush(response).addListener(ChannelFutureListener.CLOSE);

},

error -> {

FullHttpResponse timeoutResponse = new DefaultFullHttpResponse(

HttpVersion.HTTP\_1\_1,

HttpResponseStatus.NO\_CONTENT

);

ctx.writeAndFlush(timeoutResponse).addListener(ChannelFutureListener.CLOSE);

}

);

}

}

**🧪 Behavior Summary**

| **Feature** | **Implemented?** |
| --- | --- |
| Durable queue | ✅ via SimpleMessageListenerContainer |
| Persistent messages | ✅ ActiveMQ default unless overridden |
| Manual ack | ✅ using Session.CLIENT\_ACKNOWLEDGE |
| Redelivery on failure | ✅ since we don’t acknowledge() on exception |
| Backpressure handling | ✅ Flux.create with .next() |
| Timeout fallback | ✅ via .timeout(Duration) |
| Netty long-poll response | ✅ sends ack or 204 No Content |

**✅ Final Notes**

* You can safely support **multiple long-poll clients** with this design.
* You may extend this to filter ACKs by clientId or messageId (e.g., maintain per-client Flux emitters).
* For **high throughput**, consider scaling the SimpleMessageListenerContainer concurrency or batching ACKs.

Would you like this wrapped as a Spring Boot starter project for easier testing?

**You said:**

Flux.create(emitter).next().subscribe() so what is the use of calling next() method here can't we do directly subscribe()

**ChatGPT said:**

Excellent question — let’s clarify this subtle but important difference:

**🧠 Flux.create(...).next().subscribe() vs Flux.create(...).subscribe()**

They **do not behave the same way**, and which one you use depends on your intent.

**✅ 1. Flux.create(...).next().subscribe()**

**🔍 What it does:**

* .create(...) builds a potentially infinite stream (Flux<String>).
* .next() converts it into a Mono<String> — meaning:
  + Only the **first emitted item** is handled.
  + After receiving that item, it automatically cancels the subscription.

**✅ Use Case:**

* You only want **the first available message**, then stop.
* This is perfect for **long-polling**: return the first ACK, then close.

java

CopyEdit

Flux<String> ackFlux = Flux.create(emitter -> {

emitter.next("ack1");

emitter.next("ack2"); // ignored by .next()

});

ackFlux.next().subscribe(System.out::println); // Only prints: ack1

**❌ What happens if you just do .subscribe()?**

java

CopyEdit

Flux.create(emitter -> {

emitter.next("ack1");

emitter.next("ack2");

}).subscribe(System.out::println); // prints ack1 and ack2

* You now get **all events**, not just the first.
* Also, you must handle **cancellation**, **resource cleanup**, and **completion** manually.
* Not ideal for one-shot long-poll behavior.

**🧾 In Your Case (Long Poll ACK):**

You want:

* The **first ACK** from ack-queue.
* Return it via HTTP.
* Close the Netty channel.
* Do not wait for more.

✅ So this is exactly why:

java

CopyEdit

ackFluxEmitter.getAckFlux()

.next() // get only first ACK

.timeout(Duration.ofSecs(30))

.subscribe(...); // send response and close

is the **right approach**.

**✅ Bonus: What if you want to keep long-poll open for *all* ACKs?**

Then remove .next() and use .subscribe(...) directly — but remember:

* You must handle .cancel() after timeout.
* And maybe buffer or batch responses.

**✅ TL;DR**

| **Code** | **Behavior** | **Best for** |
| --- | --- | --- |
| flux.next().subscribe() | First item only, auto-cancels | ✅ Long-poll (single response) |
| flux.subscribe() | All items, ongoing | Streaming / continuous ACKs |

Let me know if you want examples of switching from .next() to full streaming (like for WebSocket or SSE).

**You said:**

where is long polling we are doing in our case actualy i am new so asking

**ChatGPT said:**

That’s a great and important question — and you're absolutely right to ask it.

Let me break it down clearly with respect to **your current architecture** and how **long polling fits in**.

**✅ What is Long Polling?**

**Long Polling** is a technique where a client sends an HTTP request to the server and waits (often for several seconds) until:

* A response becomes available, or
* A timeout occurs (e.g., 30 seconds), and the server replies with no content.

The key point: the server **does not respond immediately**; it holds the request **open** until some event happens (e.g., a new ACK).

**🧠 In Your Case – Where Is Long Polling?**

Let’s break down your flow.

**👇 You said earlier:**

After sending a message from service-ui to push-server, we send another request to wait for acknowledgement.

So you do this:

1. service-ui sends push notification to push-server via HTTP.
2. Then it sends another request to push-server — asking:

“Let me know when the target device ACKs this message.”

That second request is a **long-polling ACK request**.

**Here's what long polling looks like in code:**

java

CopyEdit

ackFluxEmitter.getAckFlux()

.next() // long-polling: wait for 1 message

.timeout(Duration.ofSeconds(30)) // wait max 30s

.subscribe(... send HTTP response back ...)

* This is **long polling**, because:
  + It doesn’t respond immediately.
  + It holds the connection open waiting for an ACK from the queue.
  + If nothing is received in 30 seconds → it times out.
  + If an ACK is received → it returns immediately.

✅ **That is long polling in action**!

**📘 Why Not WebSocket?**

* You **could** use WebSocket or Server-Sent Events for continuous ACKs.
* But long polling is **simple and works well** when:
  + You only need 1 reply at a time.
  + The client is okay with re-sending the request again after a timeout.
  + You want to avoid full-duplex state management.

**✅ Summary**

| **What** | **Your Code** |
| --- | --- |
| Long Polling? | ✅ Yes, it happens when service-ui sends an HTTP request and push-server holds it until ACK arrives. |
| How is it implemented? | Flux.create().next().timeout().subscribe() |
| Why is .next() important? | It makes it a **one-shot** long-poll instead of a continuous stream. |
| What if no ACK comes? | A 30-second timeout triggers and a 204 No Content is returned. |

Would you like a visual diagram of this whole flow (push ➝ ack ➝ long poll)? It may help you solidify how the pieces connect.

**You said:**

how are you ensuring retry incase connection was broken at the time message event was emitted so retry again same message when connection become available

**ChatGPT said:**

Great follow-up! Ensuring **reliable delivery** of ACKs (or any event) even when the consumer connection is broken is one of the **core reliability concerns** in any event-driven system.

**🔍 Your Concern**

You want to make sure:

If an ACK message is emitted (received from device and pushed to ActiveMQ), but service-ui's HTTP long poll request **was not active (disconnected, restarting, or broken)** — then that message should **not be lost**, and should be delivered as soon as the connection is back.

✅ Yes — you want **guaranteed redelivery** of missed ACKs.

**✅ How It’s Already Handled — If Done Correctly**

The **combination of ActiveMQ** + **manual acknowledgment** + **Flux-based long polling** can **guarantee** reliable delivery — but only if configured correctly.

**✅ 1. ActiveMQ Message Persistence + Durable Queue**

When the push-server sends the ACK to the ack-queue, ensure:

* The message is **persistent** (not memory-only).
* The queue is **durable** (survives restarts).
* The broker will **retain the message until a consumer acks it**.

In push-server:

java

CopyEdit

producer.send(message, DeliveryMode.PERSISTENT, priority, ttl);

So even if the service-ui is **disconnected**, the ACK message will **remain in the queue**.

**✅ 2. SimpleMessageListenerContainer in push-server or a shared component**

This uses:

java

CopyEdit

container.setSessionAcknowledgeMode(Session.CLIENT\_ACKNOWLEDGE);

This means:

* The message is **not removed** from the queue until your handler explicitly calls:

java

CopyEdit

message.acknowledge();

So if the downstream channel (e.g. HTTP connection to service-ui) **fails** before acknowledging, the message will be **re-delivered automatically**.

**✅ 3. Retry Logic is Built-in via Redelivery**

Let’s walk through what happens when a message is delivered but the HTTP channel is broken:

**✅ Scenario:**

1. ACK message is delivered to FluxSink (via ActiveMQ listener).
2. FluxSink.next(ack) is called → subscriber tries to send response via HTTP.
3. Netty HTTP connection is broken or closed unexpectedly.
4. The message.acknowledge() is **never called** due to exception.

**🧠 What happens then?**

* ActiveMQ sees: “Consumer did not ack the message.”
* It retries message delivery **after reconnect**.
* If SimpleMessageListenerContainer is still running or restarts, it receives the message again.
* If the service-ui resends the long poll request → it gets that message.

✅ This is **exactly how retry happens automatically**.

**✅ 4. Backpressure Awareness — Wait Until Client is Connected**

You use this:

java

CopyEdit

Flux.create(emitter)

.next()

.subscribe(... send response to client ...);

This guarantees:

* You wait until the client is available (connected via HTTP).
* As soon as there’s a message, you respond and close.
* If the client is disconnected and the message fails to deliver → no acknowledge() happens → **message is retried**.

**✅ Optional Enhancement: Deduplication**

Sometimes the client might accidentally receive the **same ACK twice** due to a retry.

You can include a **UUID** or **correlationId** in each ACK message:

json

CopyEdit

{

"ackId": "msg-123",

"status": "delivered"

}

Then the client (service-ui) can safely discard duplicate ACKs.

**✅ Final Summary**

| **Feature** | **Where It's Handled** |
| --- | --- |
| Message durability | DeliveryMode.PERSISTENT |
| Message retry | CLIENT\_ACKNOWLEDGE mode (manual ack) |
| Retry after disconnect | Automatic via ActiveMQ redelivery |
| Ensuring client is ready | Flux.create().next() + subscribe() |
| Deduplication | Add ackId or messageId in payload |

Would you like to see how to **simulate this retry behavior** in a test setup (e.g., disconnect service-ui, emit an ACK, then reconnect and verify redelivery)?

**You said:**

i just want to handle failure failure in case connection broken from the server-ui side for now i am not handling if push-server goes down so please give me above details based on that

**ChatGPT said:**

Perfect — thanks for the clarification.  
Let’s now **focus only on handling failure from the service-ui side**, i.e.:

**🧠 Problem Statement**

When service-ui sends a **long-polling ACK request** to push-server, and then:

* The connection is **broken** (e.g., browser refresh, network issue, or service-ui restart), or
* The request times out before an ACK arrives…

👉 You want to **ensure that no ACK message is lost** and the next long-polling request **can still receive it** when it reconnects.

✅ push-server is **always running**, so **we don’t need to handle its failure.**

**✅ Desired Outcome**

1. **ACK messages must be held** on the server side if no client is connected.
2. When service-ui **reconnects**, it should **get any pending ACK**.
3. Messages should **not be acknowledged** (and removed from queue) unless they are **successfully delivered** to service-ui.

**✅ Solution Design (based on ActiveMQ + Netty + Flux)**

Here’s exactly how to achieve it:

**✅ 1. ActiveMQ Queue Setup (Guaranteed Delivery)**

Ensure when you send ACKs (from device to ack-queue):

* Messages are persistent
* Queue is durable
* Consumer uses manual ack

**🔧 Code in push-server (Device ACK Side):**

java

CopyEdit

TextMessage message = session.createTextMessage(jsonAck);

producer.send(message, DeliveryMode.PERSISTENT); // Ensure durable storage

**✅ 2. push-server → Listen with Manual Ack and Only ack After Successful Delivery**

We’ll store the emitter temporarily for each long-polling HTTP connection.

java

CopyEdit

public class AckFluxEmitter {

private final Queue<FluxSink<String>> waitingClients = new ConcurrentLinkedQueue<>();

public void registerClient(FluxSink<String> sink) {

waitingClients.add(sink);

}

public void deliverAck(String ackJson, Message jmsMessage) {

FluxSink<String> sink = waitingClients.poll();

if (sink != null) {

try {

sink.next(ackJson); // backpressure-aware

sink.complete(); // close Flux

jmsMessage.acknowledge(); // manual ack only after successful send

} catch (Exception e) {

// if we failed to deliver (connection broken), do not ack

waitingClients.add(sink); // re-register to wait again

}

} else {

// no client connected, just wait (message remains in queue)

}

}

}

**✅ 3. Netty ChannelDuplexHandler for Long-Poll**

java

CopyEdit

@Override

public void channelRead(ChannelHandlerContext ctx, Object msg) {

if (msg instanceof FullHttpRequest req && "true".equals(req.headers().get("ack"))) {

Flux.create(emitter -> {

ackFluxEmitter.registerClient(emitter);

}).timeout(Duration.ofSeconds(30))

.subscribe(

ack -> {

FullHttpResponse res = new DefaultFullHttpResponse(

HttpVersion.HTTP\_1\_1,

HttpResponseStatus.OK,

Unpooled.copiedBuffer(ack, StandardCharsets.UTF\_8));

ctx.writeAndFlush(res).addListener(ChannelFutureListener.CLOSE);

},

err -> {

// timeout or disconnect: clean response

ctx.writeAndFlush(new DefaultFullHttpResponse(HttpVersion.HTTP\_1\_1, HttpResponseStatus.NO\_CONTENT))

.addListener(ChannelFutureListener.CLOSE);

}

);

} else {

ctx.fireChannelRead(msg);

}

}

**✅ 4. JMS Listener Hook (to emit messages to client if available)**

java

CopyEdit

container.setMessageListener(new MessageListenerAdapter(new Object() {

public void handleMessage(String ackJson, Session session, Message rawMsg) {

ackFluxEmitter.deliverAck(ackJson, rawMsg); // tries to send to any waiting client

// only calls rawMsg.acknowledge() if it successfully delivers

}

}));

**✅ Why This Works**

| **Scenario** | **What Happens** |
| --- | --- |
| service-ui makes long poll request | Emitter is registered in memory (waitingClients) |
| ACK is received from device | Message is pushed to queue, listener picks it up |
| service-ui is still connected | Message is delivered, FluxSink emits, message is acked |
| service-ui connection breaks | FluxSink write fails, message is **not acknowledged**, remains in queue |
| service-ui reconnects | New FluxSink is registered, and unacked message is retried |

This is **exactly what you want**: **durable delivery to a reconnecting client** if its connection was dropped earlier.

**✅ Summary**

| **Feature** | **Ensured** |
| --- | --- |
| Message not lost if service-ui disconnected | ✅ |
| Message redelivered on reconnect | ✅ |
| Manual ack() only after successful delivery | ✅ |
| In-memory emitters manage active clients | ✅ |
| No need to handle push-server crash | ✅ (as per your scope) |

Would you like a ready-to-run code template or test case for simulating this behavior?