Design and Implementation of a Pub-Sub System

Sami Hatoum

1 Overview

Publish-and-Subscribe messaging-passing systems involve two parties which distribute events on a topic across an asynchronous channel. It is up to the middleware to process the events from publishers such that events passed forward are all present, in the correct order, and with proper error handling, ensuring little reliance placed on either party. A queue-centric approach is best suited to handling messages, as it can ensure ordering (FIFO), and can also be blocking, only handling data when able to. This practical will discuss a competing-consumers implementation, one which divides the messages equally (or close to) amongst subscribers rather than broadcasting; this will also detail how it has been tested, and how it can be run.

2 Design and Implementation

The breakdown of this implementation can be divided into the primary functionality requirements of the PubSub system, or more accurately, the Consumer-Competition system (R1 - R3), followed by all reliability/fault-control considerations to ensure this full and complete transmission of events across the middleware (R4 - R10). This practical is built on the basis of the Java Remote Message Interface (RMI). This library is designed for remote method calls on objects located elsewhere on the network (the localhost), as if they were local objects, abstracting the network complexity involved in communicating between separate JVM processes. In effect, this facilitates the communication between the publishers, the broker, and the subscribers.

2.1 R1: Subscribing and Publishing Events

First, for publishing and subscribing events, an event channel is hosted on a server (as a remote object, a "stub") and bound to the RMI registry. The client (publisher or subscriber) retrieves the channel's stub from the registry by name; invoking their associated functions on the stub as if it were local. The stub forwards this call over the network to the server side which delivers it to the actual event channel object, thus doing all the actual processing. The full process is detailed in Figure 1. The way events are consumed is via a push-mode strategy, in which the subscriber component provides an interface that the channel can call when an event is available (i.e., the middleware determines when a subscriber is processing an event). To avoid subscriber clashes, it is assumed the ID of each client is unique.

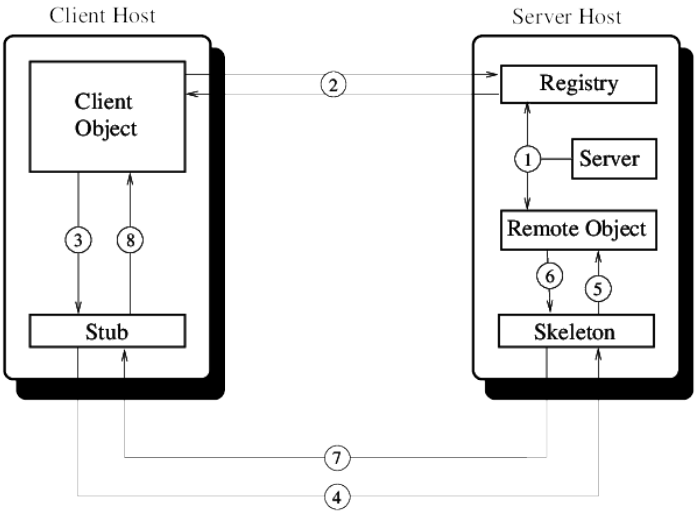

Figure 1: Java RMI Architecture. Source: [2]

2.2 R2: Lookup/Discovery & Access of Channels

Event channels are bound to the RMI registry. Simply, knowing the name of the channel allows it to be accessed by any clients, or alternatively, the available channels can be listed out (via the built-in functionality of the broker) for the clients to interact with.

2.3 R3: Dynamic Queue Management

Event channels revolve entirely around dynamic creation and manipulation of queues. As described in R1, channels provide essential publish/subscribe functionality. Publishers insert messages into a global priority queue, which is then segmented into sub-queues (with priority ordering) for equitable data distribution amongst subscribers. Each message includes some metadata, including a "partition ID" and timestamp, to facilitate the sequential organisation of messages, ensuring correct ordering during asynchronous data distribution. The partition ID enables publishers to group events into distinct blocks. When a block is sent to the channel, it is allocated entirely to a single subscriber rather than divided amongst multiple subscribers. This method ensures the correct sequence of processing within each data block, independent of subscriber processing speed variations. The timestamps are used for sorting events in the queues, to facilitate in-order message processing within the blocks themselves. This is a more robust system than simply relying on inherent FIFO queue properties. Additionally, real world scenarios would employ synchronised clocks to ensure timestamps are consistent across all publications—we assume the same for this system (for testing purposes, this holds, as all functionality is hosted locally).

For proper load balancing, the "Kafka model" is used, detailed by Comartin [1]. When scaling to multiple publishers, all events initially enter the global queue. Each partition, uniquely identified by a combination of publisher ID and partition ID, is assigned its own dedicated queue. Subscribers are then allocated complete partitions, with each partition processed exclusively by one subscriber. However, individual subscribers may simultaneously handle multiple partitions. This strategy is illustrated in Figure 2 and is the closest solution I was able to implement which could guarantee FIFO delivery and FIFO processing, though there is a slight reliance on the publisher for extra information. In one potential scenario, a partition can be disproportionately large compared to other partitions, thus making the work of one subscriber much greater than the others; this will only occur if there a small number of events to be processed outside of this block of data. The issue presented is a trade-off that inherently exists in a competing-consumer model. My attempt to combat it is through an estimation of how much processing a subscriber is currently handling—the subscriber with the least number of events in its queue will be selected next to process an event. This calculation also weighs the number of partitions a subscriber is handling more highly, as these cannot be interchanged between subscribers unless the subscriber is inactive or processing them too slowly. Tracking the allocation of subscribers to partitions is accomplished by employing several concurrent hashmaps. These data structures systematically record the associations between partitions and subscribers while also mapping unique identifiers to their corresponding queues. This ensures an organised and efficient representation of the dynamic structures within the system. Concurrent structures will ensure multiple threads accessing the same data will not override one another.

As the entirety of this distributed system is built locally, individual channels, and subscribers on those channels, are separated into threads (and queues) so they can run simultaneously from an single class instance. Threads are coordinated in this system around a shared condition: the availability of subscribers. Specifically, the partition-offloader thread uses a synchronised block to safely call `subscriberLock.wait()`, temporarily pausing its execution when no active subscribers are available, and releasing the lock, to allow other threads (like subscriber registration threads) to proceed. Similarly, when a new subscriber registers, the subscribe method enters a synchronised block to safely call `subscriberLock.notifyAll()`, waking up any waiting threads to re-check subscriber availability. This is crucial to ensure consistent state when accessing shared resources like the subscriber map.

One final consideration for dynamic queue management is the limit on the number of events which could occupy a single queue at a time. For the purposes of demonstrating this systems functionality, the limit has been set to 1000 messages, though even this is excessive as events are processed near-instantaneously, and thus will not remain in a single queue long enough for this threshold to be hit. There is no acknowledgment between the publisher and the channel, so rejected messages will simply not be accepted into the queue.

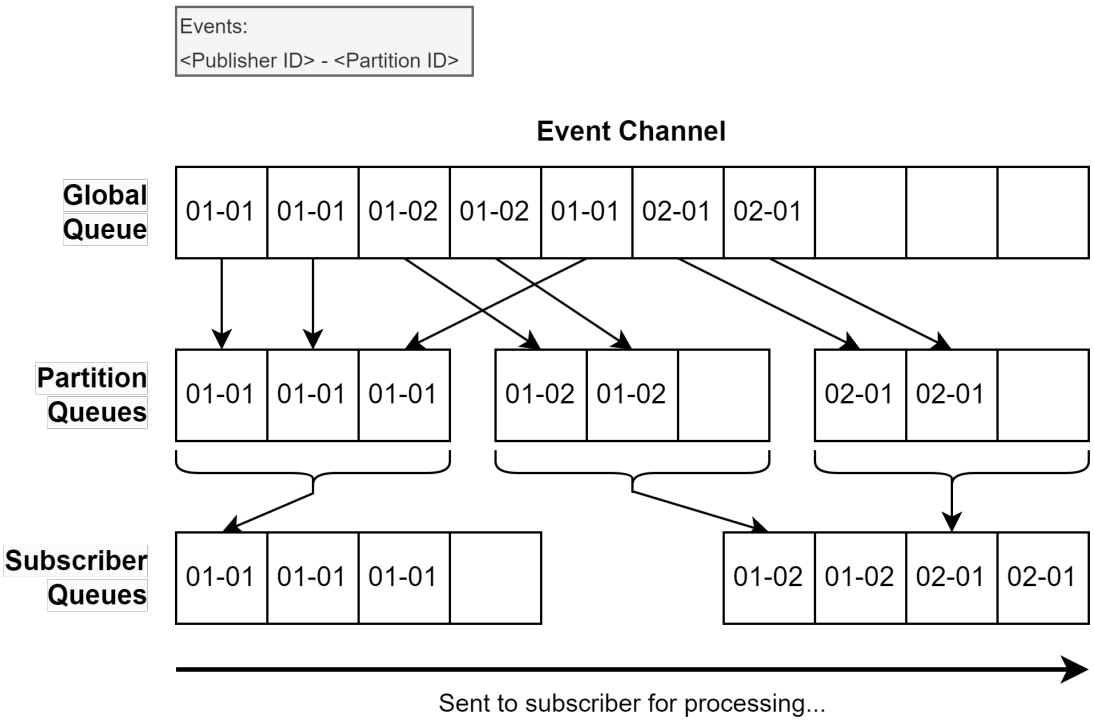

Figure 2: Event Channel Dynamic Queue Management

2.4 R4: Temporary Interruptions of Connections

Due to my own uncertainty about what a temporary interruption of connection consisted off, I considered two definitions, each handled in a different way. The first idea was that an interrupt in the connection can be considered as any non-instantaneous processing of data by the subscriber. This is simulated in the code with a 300ms delay, with a 15% chance of occurring (due simulate fluctuations in connection). Interruptions to connections here are not punished; there is no scenario where too many minor time delays will cause the subscriber's events to be redistributed or the subscriber disconnected. Merely, these are recognised by the code (and printed out), however they are not considered detrimental enough for further consideration.

The alternative was that a temporary interrupt could be caused when the subscriber disconnects from the channel and reconnects (i.e., closing the subscriber client and opening it up again). In this instance, the subscribers enter a safety 'shut-down hook', which accounts for these crashes and will unsubscribe the subscribers from all channels they are currently in. The effect of this is all events in all the subscribers queues are redistributed back to their partition queues where they can be divided amongst the subscribers again, once they reconnect. Both have been implemented.

2.5 R5: Crashing Queues

Recovery of crashing queues makes use of a backup global queue. For each event published, the data is sent to both primary queues; only when a subscriber confirms it has received an event, can the corresponding item be removed from the backup queue, as it has been processed and will not be required again. The reason this implementation holds is due to the priority (by their timestamps) transmission of data, such that if the subscriber received a message, all subsequent messages of the partition would be ones later in the sequence, and so only those would need to be retransmitted by the publisher. Rather than having a backup queue for each current active queue, only the global queue is given a backup. Assuming all hashmaps are retained during a queue crash, this would allow all partitions to be reassigned immediately, without the significant increase in space required to store copies of every queue.

Upon the queues crashing, all publishing actions are momentarily halted, while the global queue is repopulated with the contents of the backup queue in the correct order, and the backup queue wiped. This blocking restoration process is indicated by a flag, which is passively checked for in the main thread when partitioning events. If there is a difference between the number of messages published and delivered, and there is no backlog of messages in the queues, it means the queues have crashed. This assumes that the queues crashing does not crash the backup queue as well. Implementing this is simulated via a manual function call by the publisher, emptying all the queues.

2.6 R6: Crashing Customers

Simulating a crashing customer (i.e., the message has been sent to a subscriber who is not active) can be identified by the channel via strategic use of the RMI remote exception. My implementation ensures the only time this exception will be thrown during subscriber processing is if the subscriber has 'crashed'. In this instance, its work will be offloaded to the other subscribers in the queue, which can be achieved by ending its thread; the offloader thread will detect this change and redistribute its events to other available subscribers, removing all references to the inactive subscriber from hashmaps in the process. This extends to even the current partition being processed; the remaining events in the queue for that partition will also be reassigned when a crash occurs. If there are no available subscribers to be reassigned to, the synchronisation lock will be enabled, halting all threads until a subscriber (re-)connects.

Extra consideration has been given to the current partition being processed. The definition of in-order processing matters here—this implementation defines it as: "messages of a partition must be processed in order, by any subscriber". Typically, this means that sending the complete block to one subscriber will handle this well. However in a crash, reassigning only the unprocessed messages to another subscriber, instead of the whole block again, will suffice, as ordering is maintained.

Subscribers are given three attempts resending the message before they are considered inactive, at which point their contents are moved to another subscriber.

2.7 R7: Long Delays in Network Traffic

The distinction given between a long delay in network traffic, and (the first definition of) a minor interruption largely revolves around the duration of the delay. This can come in two forms: either a single message is taking long to process, or a subscriber is experiencing many (near-)consecutive delays in its processing. The latter of these is more important, in these instances, regardless of how many partitions the subscriber is currently processing, it will need some of its work to be offloaded to other subscribers. Each subscriber has its own set of counters and timers related to the duration of the delays; if this is exceeded frequently in a short period of time, its events are distributed to other subscribers. The subscriber will however still be active, so it will be able to receive new events, in the hopes the network delays have ended. Doing so will ensure that one subscriber will not be disproportionately over-worked when it is not able to handle it all.

2.8 R8: Dropped Messages

Dropping messages are handled entirely between the channel and its subscribers. Checking for dropped messages employs a minor acknowledgment system, built into the receive functionality. A message will be resent to the subscriber until the receive function is called successfully, flagging the acknowledgment as true. The channel will continue attempting to send the message to the subscriber until the acknowledges the message or the number of attempts runs out. If these checks fail, much like the crashing customers have been handled, the contents of the subscriber are redistributed for others to process, and the subscriber is marked as inactive.

2.9 R9: Out of Order Messages

Out of order messaging is a property inherently tackled by the implementation of dynamic priority queueing, specifically the use of partition queues. These maintain the ordering in which a block of events are processed (by offloading them all to a single subscriber), and the sorting of queues by timestamp allows them to be processed in the order the publisher intended. Additionally, the backup queue maintains the correct ordering when repopulating the queue for the same reason—priority ordering.

2.10 R10: Duplicated Messages

Two approaches can be taken for handling duplicated messages, whether it is allowing the same message to be sent more than once, or the same message to be delivered more than once, I chose the latter. Here, if a subscriber has received the message and gave acknowledgment, the message's unique ID is added to a set of all delivered IDs. If the message were to be delivered again for any reason, it would be discarded by the channel. The methodology follows an "at-least once" delivery; every message is guaranteed to be delivered at least one time because it is resent to the subscriber until acknowledgment is given for delivery.

3 Evaluation

The evaluation will cover testing of each requirement, ensuring their full functionality before other requirements were built on top of existing code. Aside from ensuring the channels can run simultaneously (as part of R3), all tests will be done on a single channel, as running multiple concurrently will not change the outcomes; this statement has been tested as well, though will not be included in the tables below. Testing is divided into: the test case, including its requirements, followed by the expected outcome and whether or not is has passed the test.

Prerequisite	Test	Expected Outcome	Pass/Fail
N/A	Testing publishing first: Publishing events to a channel, no subscribers	Messages will be published but the offload will wait until a subscriber is ready	Passed
	Testing subscribing first: Publishing events to a channel with two available subscribers	Messages will be published and distributed equally between the subscribers	Passed
	Testing queue limits: Publishing 1000 messages to a channel. 2 subscribers and 2 partitions of data	All messages will be received within 1 second and partitioned correctly	Passed
	Testing overloading global queue: Publishing 10000 messages to a channel without removing from the global queue	The first 1000 messages will be stored in the queue, the remaining 9000 are dropped	Passed

Table 1: **R1: Publishing and Subscribing** Test Cases

Prerequisite	Test	Expected Outcome	Pass/Fail
N/A	Testing dynamic channel lookup: Create two channels on broker startup, and then a third by a publisher	Two channels displayed in the console, then all three after the new channel is made	Passed

Table 2: **R2: Channel Lookup & Discovery** Test Cases

Prerequisite	Test	Expected Outcome	Pass/Fail
N/A	Testing large partitions: Publish 10 messages in 2 partitions, with 2 available subscribers	The two partitions are distributed equally to the 2 subscribers, creating, populating, and emptying all queues as events are processed	Passed
	Testing small partitions: Publish 5 messages in 5 partitions, with 5 available subscribers	Each subscriber will receive a partition containing one message	Passed
	Testing multiple active channels: 5 messages in 5 partitions, with 5 available subscribers. Done simultaneously between 2 channels	Each subscriber will receive a partition containing one message	Passed

Table 3: **R3: Dynamic Queue Management** Test Cases

Prerequisite	Test	Expected Outcome	Pass/Fail
Change the subscriber's `receive()` function to `receiveDelayed()`	Testing the indicator for a small delay: Publish 10 messages, 2 partitions for 2 subscribers. A small delay has a simulated 15% of occurring	1-2 messages will trigger the small-delay flag, printing an indicator	Pass
	Testing the redistribution of messages after closing the subscriber client: Publish 10 messages, 2 partitions for 2 subscribers. One subscriber will fully process its messages and the other will process one. The client is then closed	The remaining 4 messages to be processed by the second subscriber are instead put back into their original partition queue and whichever subscriber reconnects first will process all 4 messages.	Pass

Table 4: **R4: Temporary Interruptions of Connection** Test Cases

Prerequisite	Test	Expected Outcome	Pass/Fail
Call `channel.killQueues()` from the Publisher	Testing backup recovery after crashing the queues: Publish (but don't process) 10 messages. At the 5th message simulate crashing the queues	The first 5 messages are delivered and the latter 5 are rejected. The backup will repopulate the global queue with the first 5 so they can be sent to the subscriber	Passed
	Testing backup clearing after crashing the queues: Publish and process 10 messages. At the 5th message simulate crashing the queues	The first 5 messages are delivered and the latter 5 are rejected. The backup will be empty, as the first 5 messages have been processed by a subscriber, so nothing is needed in the backup	Passed

Table 5: **R5: Crashing Queues** Test Cases

Prerequisite	Test	Expected Outcome	Pass/Fail
Change the subscriber's `receive()` function to `receiveCrashed()`	Publish 10 messages, 2 partitions for 2 subscribers. The second subscriber will have a deliberate crash after the 4th (out of 5) message	The first 3 messages are delivered successfully, the 4th will crash, leading to the 4th and 5th messages being passed to the other subscriber	Passed

Table 6: **R6: Crashing Customers** Test Cases

Prerequisite	Test	Expected Outcome	Pass/Fail
Change the subscriber's `receive()` function to `receiveDelayed()`	Testing event redistribution after too many long delays: Publish 10 messages in 2 partitions, with 2 available subscribers. A long delay will have a 15% chance of occurring	After 3 delayed messages, the 2 unprocessed messages in the queue, will be passed to the other subscriber, who is experiencing no delays	Passed

Table 7: **R7: Long Delays in Network Traffic** Test Cases

Prerequisite	Test	Expected Outcome	Pass/Fail
N/A	Testing retry attempts on receiving messages: Publish 10 messages in 2 partitions, with 2 available subscribers. Only one subscriber will fail to acknowledge some of its messages	After the third attempt sending a message to the subscriber, it is marked as inactive. The current message, and the remaining messages in the queue are redistributed	Passed

Table 8: **R8: Out of Order Messages** Test Cases

Prerequisite	Test	Expected Outcome	Pass/Fail
N/A	Testing in-order processing. Publish 10 messages in 2 partitions, with 2 available subscribers	The two partitions are distributed equally to the 2 subscribers. Each message is processed in chronological order (determined by the timestamps). Each block is processed by a single subscriber, unless for error conditions details in the other test tables	Passed

Table 9: **R9: Out of Order Messages** Test Cases

Prerequisite	Test	Expected Outcome	Pass/Fail
N/A	Testing the delivered messages set. Publish 10 messages in 2 partitions, with 2 available subscribers. Do it again with the same 10 messages	All messages in the second round of publications will simply be dropped, as they have already been delivered to a subscriber	Passed
	Testing the delivered messages set. Publish 10 messages in 2 partitions, with 2 available subscribers. Do it again with one new message	All messages in the second round of publications will be dropped aside from the one unseen message	Passed
	Testing the delivered messages set after disconnecting subscribers. Publish 10 messages in 2 partitions, with 2 available subscribers. Unsubscribe all subscribers, and do it again	All messages in the second round of publications will simply be dropped, as they have already been delivered to a subscriber	Passed

Table 10: **R10: Duplicated Messages** Test Cases

4 Conclusion

The development of this PubSub system has been implemented to the full extent of the specification, capable of publishing and receiving messages asynchronously across multiple live channels hosted over a middleware server. This come with robust error handling for tackling simulations of real world firmware failure and disruptions in the connections.

However, there is added Quality of Service (QoS) I would have wished to add to the system if given more time. The first of which is a more flexible connection order to the channels. In the current implementation, subscribers must connect before publishers due to processing being so fast that subsequent subscribers (after the first connection) are unable to be registered in time before the full work of any queued events is immediately offloaded to the first subscriber. Solutions which would force threads to momentarily halt proved inconvenient, and solutions which would redistribute the work of active subscribers to others waiting for work would result in high added complexity in regards to maintaining in-order processing of blocks. I would have also wished to add some element of garbage collection to the channel, such that unused queues/hashmap elements are disposed off after a certain amount of time (if for instance the partition is no longer needed), however I could not find a practical way to implement this, or one which would properly simulate real world conditions. This would also extend to channels as a whole which are not being used. The last change I would make is perhaps finding a more convenient approach to running the code, rather than needing both clients and the server code active at once.

5 Instructions

To run a demonstration of the code, three classes must be running simultaneously: The Broker Server, The Subscriber Client, and The Publisher Client.

This will simulate the publication and distribution of messages across 2 channels, to 2 subscribers. This will be a total of 10 messages to each channel, however this can be modified within the Publisher Client code.

Compile and run Broker Server

```
$ ./compile.sh
```

Run Subscriber Client

```
$ ./run_subscriber.sh
```

Run Publisher Client

```
$ ./run_publisher.sh
```

References

[1] Derek Comartin. *Message Ordering in Pub/Sub or Queues*. May 2022. URL: `https://codeopinion.com/message-ordering-in-pub-sub-or-queues/`.

[2] *Java RMI Performance and Object Model Interoperability: Experiments with Java/HPC++ Distributed Components - Scientific Figure on ResearchGate*. https://www.researchgate.net/figure/Java-RMI-architecture_fig1_233943469. [accessed 9 Apr 2025].