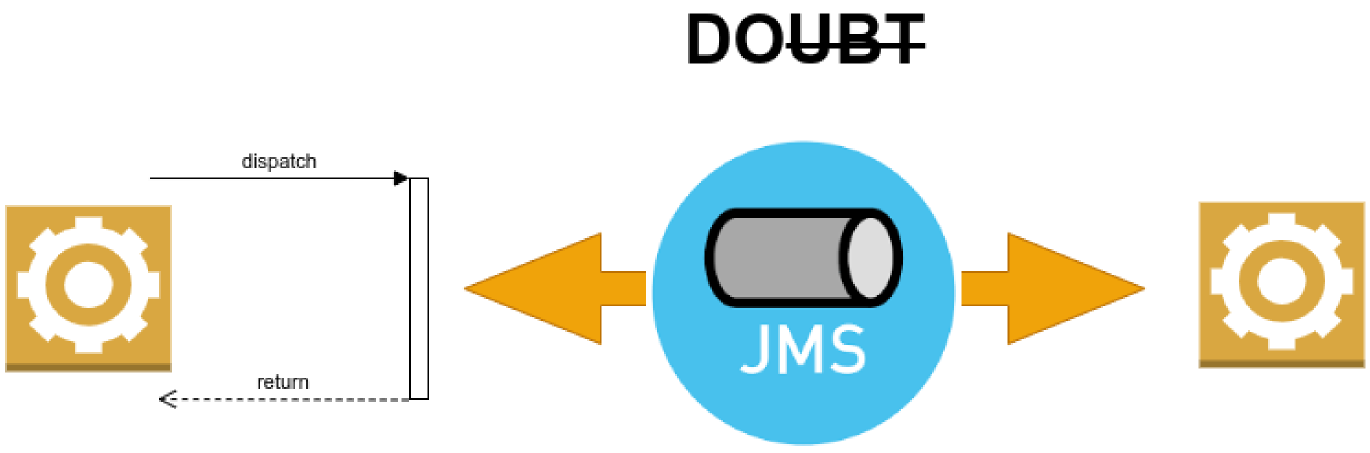
Tricky messaging, part one: synchronous transport and not only

Victor Alekseev Follow

Jun 21 · 18 min read



"The more I see of men,
the more I like dogs"

Thanks to the widespread use of Kafka, message-driven architecture has become commonplace and is widely used in a wide variety of applications. It is, of course, a considerable difference from the time of the widely known "Patterns and Best Practices for Enterprise Integration" book, when event handling in the 99% of the cases involved the tedious and bloody work of integrating multiple legacy applications.

Nevertheless, the practical techniques for organizing data processing and services interaction based on messages continue to be rather dull and primitive. In this article series, I would like to demonstrate some non-trivial tricks that simplify the application architecture quite substantially through relatively simple message handling patterns.

I plan to look at four main areas:

- Synchronous interaction (this part)
- **High-performance business processing**
- Cloud deployment with minimal overhead
- User interface

In doing so, I mainly rely on my own experience with JMS (mostly ActiveMQ for more than ten years). Also, I hope, from time to time, to draw attention to the applicability of Kafka in this context.

The last thing I want to mention before starting is the boundaries of applicability.

- If you want to compete with Amazon or deal with high-rate trading, these approaches and tools are probably not for you. It is also not worth over-complicating the system if you plan to have a couple of a dozen employees handling 100 documents a day.
- Imho, it makes sense to follow this bloody enterprise path if you have about 1–10 thousand users working on the system simultaneously, generating about several million big enough (about 10–100 kilobytes in size) documents per day. So we are talking about a typical remote banking system or other similar financial workflows.
- However, a similar architecture was once successfully applied in the completely different business area to manage a cloud of servers processing video streams and interconnect them with administrative backends and payments processing.

Introduction

It is redundant to argue that any synchronous interaction in a distributed system is a forced evil:

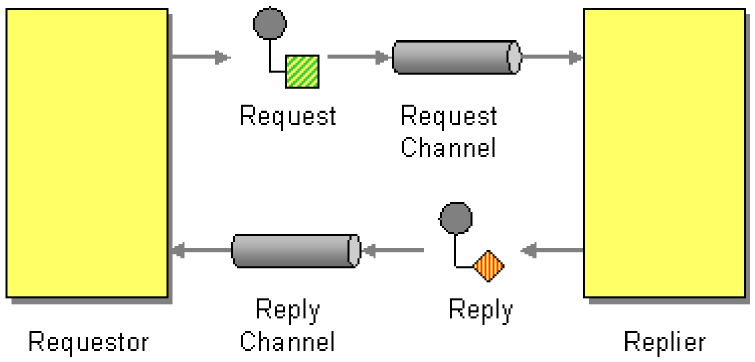
- Firstly, it requires constant availability of the both sides.

- Secondly, it blocks the calling thread. It increases the number of application threads involved and the memory and CPU overhead to switch between them.
- Thirdly, if the client takes a long time to read the result, the same threading problems occur at the called service's side.

Nevertheless, from time to time, we cannot work without this type of interaction, especially when it comes to handling the GUI and similar services on the border of the system. So let's try to mitigate the above disadvantages as much as possible.

The basis for implementation

First of all, let's remember a rather well-known pattern, "Request-Reply":



In this case, we already have "request" from the client and "reply" from the server. But don't have yet any way to wait and consume the "reply" on the calling side. To achieve that, we can apply the following approach that would significantly mitigate the strict requirements for the continued availability of the called service.

- Each client has a unique ID and subscribed for receiving messages with the filter **DestinationID = <CLIENT_ID>**

The clients' thread:

- creates the message with headers **ID = <Some UUID> and SourceID = <CLIENT_ID>**

- adds into the global collection some active request wrapper with the next properties: message UUID, and the empty storage for the future result
- sends the message to the server
- calls **wait(timeout)** on the wrapper and becomes blocked

The server:

- receives the message and processes it
- sends response with headers **CorrelationID=<request message ID> and DestinationID = <request message Source ID>**

JMS Listener on the client's side:

- receives the stream of responses from the server
- for each response:

a) using the value of **CorrelationID** header, looks for the wrapper inside the collection

b) transfers the response data from the message into the wrappers storage

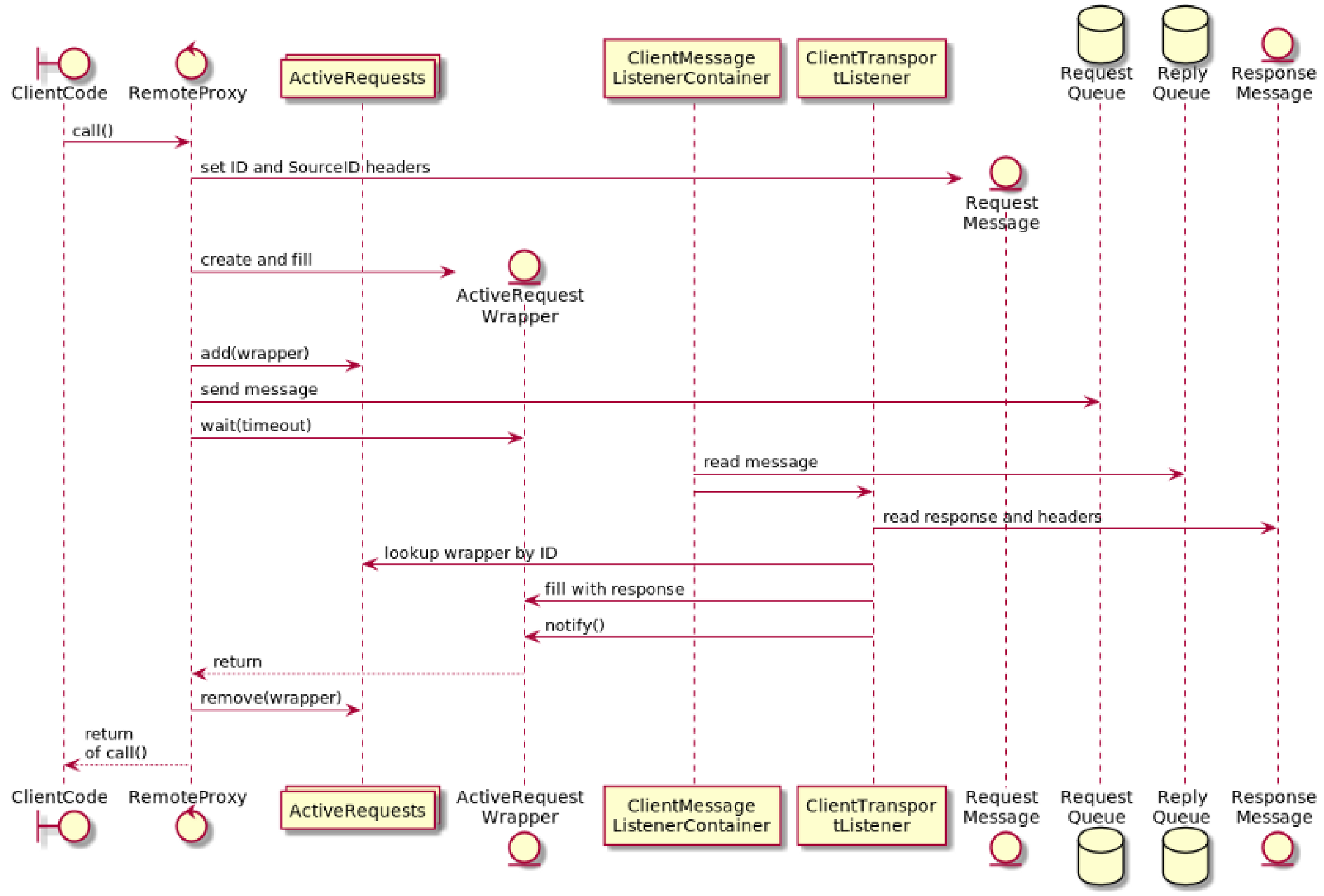
c) notifies clients thread, which is blocked on the wrapper by **wait(timeout)**

The released client's thread continues execution and:

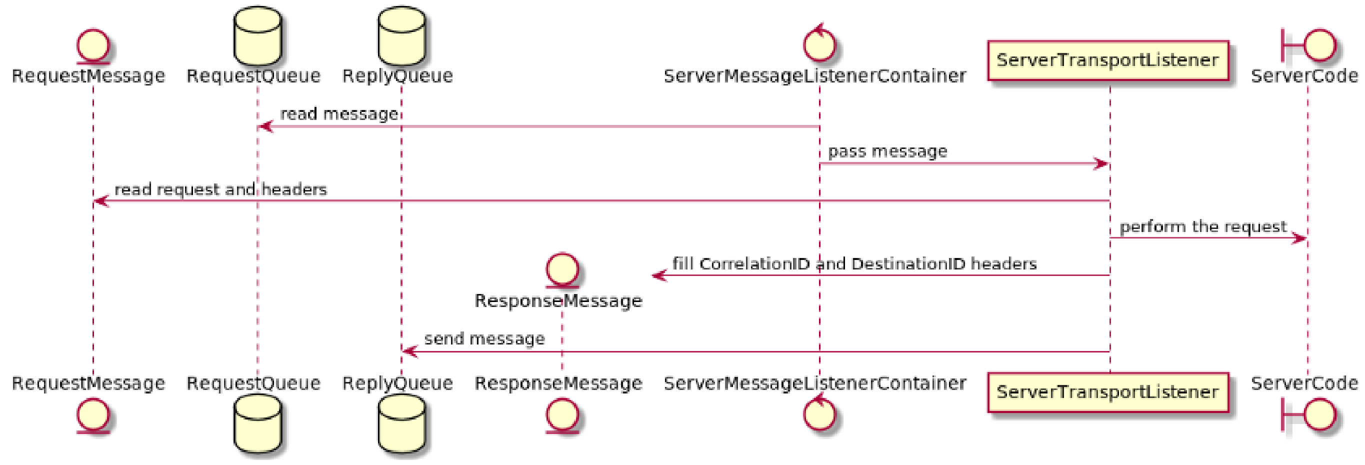
- reads the result of the request from the wrapper
- removes the wrapper from the collection
- continues normal code execution and returns the result to the calling code

Usually, from a client perspective, the details of the implementation of this approach are hidden behind a 'remote proxy' component. This component has to generated based on the business interface by the transport infrastructure. The following diagrams can illustrate the above sequence of activities:

Client's part:



Server's part:



The main benefits we have gained from this "weakened synchronous" approach:

- the server service may be temporarily unavailable (the clint too, but without ability to restart the application)
- the called thread is only blocked for the duration of the business requests execution

- server-side scalability, fault tolerance, and load balancing by increasing the number of server instances or the count of threads behind the queue listeners.
- At any time, servers may dye, shut down or restart without any problems for clients for performing upgrade, maintenance, and so on. We can upgrade the whole system gradually, one or several servers at a time, without a “maintenance window.”
- the average number of messages in the queues can easily demonstrate the system performance

Some trivial improvements, which can be useful when something goes wrong (by the way, “something goes wrong” in a distributed application happens almost all the time):

- When, using the **CorrelationID** value, we did not find a corresponding wrapper in the collection, it means that **wait(timeout)** already threw a timeout exception. In this case, it maybe makes sense to send the received response message back to the server with the command “undo the previous operation”/
- If we have organized the exchange between the stateful web server and the stateless application server similarly, it makes sense to set the TTL of the message slightly shorter than the timeout of the HTTP session. In doing so, we will not waste application server and broker resources on processing messages that no one else will need.
- The better server’s utilization can be achieved by tuning the size of the server thread pool allocated for processing incoming messages. In principle, by usage statistics on request execution times, this parameter can be automatically optimized. There are several dedicated libraries for this, for example, “Netflix Adaptive Concurrency Limits”.
  This approach is well described in the following article Performance Under Load.
  The examples to this library are designed to manage TCP connections but can easily be extended to tuning JMS message-consuming streams.
- If we make the blocking time of the execution thread much shorter than the HTTP session timeout, we have a chance to send a message to another server in case of a timeout exception. In this case, we have to take care of the server’s idempotency

because not only the request but also the response too may be lost during the communication.

The main limitation of this approach is that JMS does not guarantee the relative order of two messages close in time. In the case of a web application, for example, after sending the message, the interface has to be blocked until a response is received. Based on my experience developing complex document management applications in the financial sector, I would argue that this is not too limiting.

We'll also have to manually implement the gathering and publication of all sorts of statistics on the number of requests, response time distribution, splitting these indicators by types of requests, etc.

Advanced features

The interaction scheme described above works well but is not without some drawbacks caused by the fact that client messages are evenly distributed across all servers:

- If we use a second layer JPA cache, all reference data of all users (organizations, accounts, access rights and so on) are loaded into all server caches. In this way, the cache is used inefficiently, as the data requested usually is missing due to the lack of the server's memory and loaded each time from the database.
- If we have one or more slow or dying servers, they potentially slow down all clients.

The possible solution is "weak JMS sticky sessions". The following description is based on the already used sample of interaction between stateful web servers and stateless application servers.

- Each application server (APP) has an unique ID and subscribed for receiving messages with the filter **DestinationID = <SERVER_ID> OR DestinationID = NULL**
- Each webserver (WEB) stores attribute **ORIGIN** (its initial value is null) in the user's session and includes it as header **DestinationID** in each request to the server.

- The first message during any client's session is sent by the WEB server with header **DestinationID = null** and will be received by a random APP server. This APP server answers with header **ORIGIN = <SERVER_ID>**
- After that,the web server reads the header **ORIGIN** from the answer and stores it in the user's session

So, the following messages from this user's session will be sent by the WEB server with header **DestinationID=<ORIGIN>** and be received by the same APP server

The APP server may at any time die, restart or be slow. To prevent the "steaked" session from hanging up:

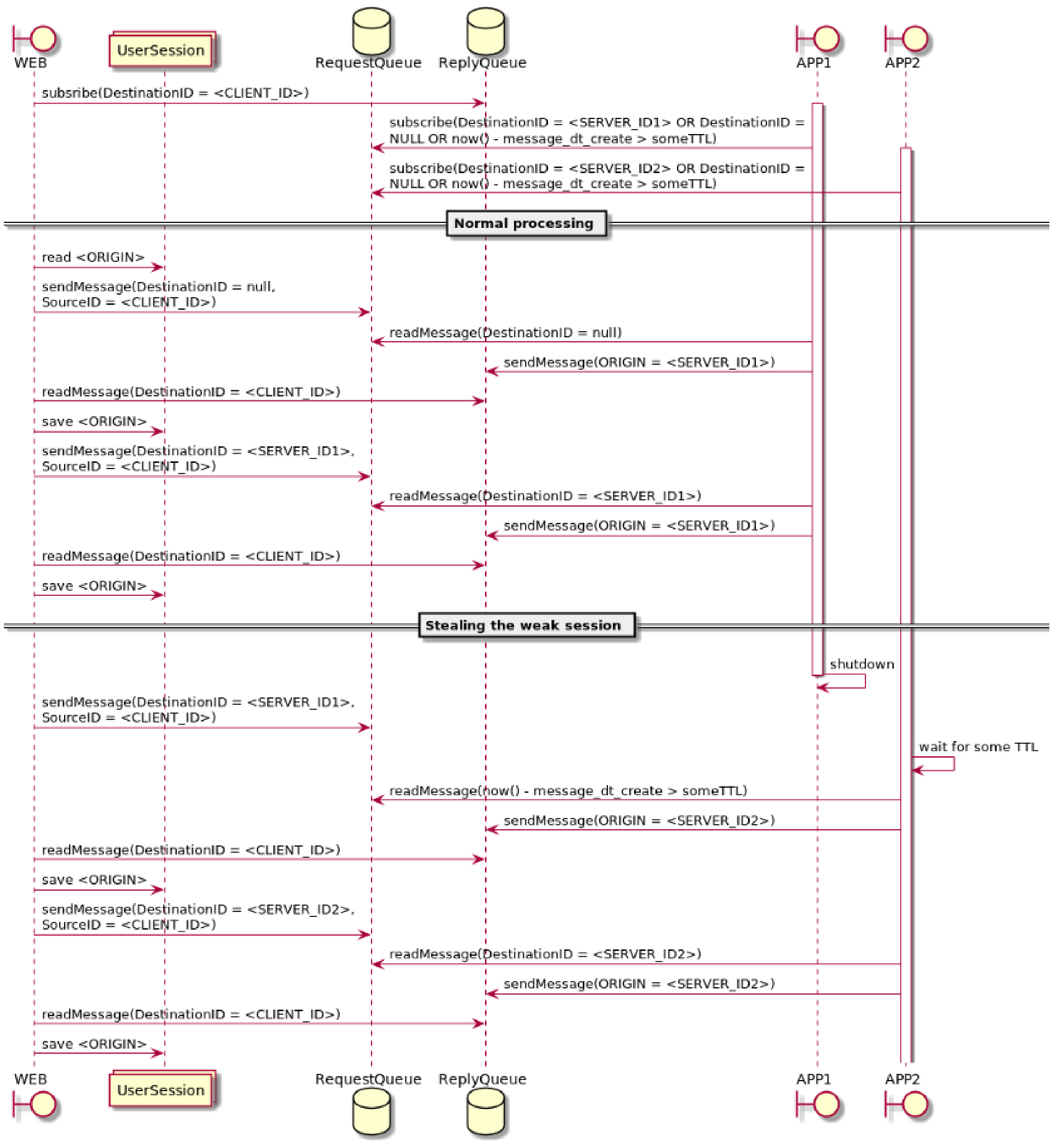
- Each APP server also subscribed for receiving messages with filter **now() — message_dt_create > someTTL**. Such a "moving" subscription can be best organized by a separate instance of **ServerMessageListenerContainer**, as far as it is periodically restarted to update the filter definition, which cannot use functions like **now()**.
- The APP server answers with one own **ORIGIN = <SERVER_ID>**, which the WEB server will store in the user's session and use by the future requestst
- So, all further messages from this user will be processed by the new APP server.
- In this case again, it is essential that only one message from each user session is in progress at a time. Otherwise, different WEB servers may continuously steal different messages, and this oscillating process of switching sessions between them could be potentially endless.

Thus, by slightly complicating the scheme, we have achieved the following benefits:

- All messages from one user are handled "if possible" by the same APP server — second level JPA cache of APP server is utilized effectively and doesn't try to load all available dictionary data
- If the APP server is slow or dead, this "weak session" will be stolen by another APP server. In this situation, we significantly disrupt the order in which the messages are processed in the queue. So we need to be completely sure that they cannot in

principle, belong to the same aggregate (e.g., a document in the course of accounting processing or a user session)

The following diagrams can illustrate the above sequence of activities:


WEB
UserSession
RequestQueue
ReplyQueue
APP1
APP2
subsribe(DestinationID = <CLIENT_ID>)
subscribe(DestinationID = <SERVER_ID1> OR DestinationID = NULL OR now() - message_dt_create > someTTL)
subscribe(DestinationID = <SERVER_ID2> OR DestinationID = NULL OR now() - message_dt_create > someTTL)
Normal processing
read <ORIGIN>
sendMessage(DestinationID = null, SourceID = <CLIENT_ID>)
readMessage(DestinationID = null)
sendMessage(ORIGIN = <SERVER_ID1>)
readMessage(DestinationID = <CLIENT_ID>)
save <ORIGIN>
sendMessage(DestinationID = <SERVER_ID1>, SourceID = <CLIENT_ID>)
readMessage(DestinationID = <SERVER_ID1>)
sendMessage(ORIGIN = <SERVER_ID1>)
readMessage(DestinationID = <CLIENT_ID>)
save <ORIGIN>
Stealing the weak session
shutdown
sendMessage(DestinationID = <SERVER_ID1>, SourceID = <CLIENT_ID>)
wait for some TTL
readMessage(now() - message_dt_create > someTTL)
sendMessage(ORIGIN = <SERVER_ID2>)
readMessage(DestinationID = <CLIENT_ID>)
save <ORIGIN>
sendMessage(DestinationID = <SERVER_ID2>, SourceID = <CLIENT_ID>)
readMessage(DestinationID = <SERVER_ID2>)
sendMessage(ORIGIN = <SERVER_ID2>)
readMessage(DestinationID = <CLIENT_ID>)
save <ORIGIN>
WEB
UserSession
RequestQueue
ReplyQueue
APP1
APP2

Some possible improvements:

- when the APP server steals the message from another “slow” server, it can tell the cluster’s infrastructure that the “slow” server has to be restarted or the size of its processing thread pool has to be reduced
- When we start a new service instance in response to a significant load growth, the number of threads configured for the “stealing” listener container can be deliberately increased to fix the accumulated mass of near-retired messages asap. Further, as the situation improves, the number of these threads can be reduced to some standard value.

In principle, the optional part of the JMS 2.0 standard provides a similar mechanism (minus the stealing of outdated messages) based on the standard **JMSXGroupId** header: “when the JMSXGroupId property is set, the JMS provider will look for a consumer that has that group ID assigned to it.
If no consumers are assigned to that group, the JMS provider will pick one based on its load-balancing scheme and assign it the group ID.
From that point on, only that consumer will receive the messages associated with that group”.
In practice, whether this feature is implemented correctly depends on the broker.For example, ActiveMQ implements it, while MQSeries (WebSphere MQ) does not.
In my personal opinion, ActiveMQ is a powerful and robust enough basis for the vast majority of enterprise projects. However, if a cross-platform capability is required at this point, or if the customer insists on using a solution from a similar respected vendor, then this behavior will have to be implemented independently.

Possible system design

Let’s briefly describe a possible application-level transport system design based on this approach. It includes the following main components:

TransportEvent — some abstraction of transport calls and results with the next main properties:

- **id, clientId, destinationId, correlationId**

- **initialTimestamp** or **expirationTimestamp**
- **List<byte[]> and List<String>** — data and types of arguments and return value
- **exceptionMessage, exceptionCode, List<String> exceptionStack**
- **destinationBeanName** and **destinationMethodName** — only for requests, are used on the server side

TransportException — general runtime exception with messages about an issue on the called side. It must contain two stack traces — one local and one remote.

Transport — abstraction of general purposes transport

- its main method is **TransportEvent send(TransportEvent event) throws TransportException**. Also we can implement and additional method for asynchronous communication — **void send(TransportEvent event) throws TransportException**
- contains replaceable strategy **StatisticsTuner** for runtime tuning containers settings.

Several implementations if **Transport,** for example00

- **JmsTransport**
- **HttpTransport**
- **LocalTransport** — for direct calling methods on beans, located in the same JVM. It can be used for testing and deployment cases when several services are started inside one monolith

Thus, by introducing similar transport abstraction, we can partially defer the decision on the principle architecture of system deployment (monolith vs. distributed) to later stages of implementation, when requirements and constraints become more apparent to developers.

Inside the **JmsTransport** where are the next components

- Two inheritors of the **DefaultMessageListenerContainer**: **MainMLContainer** for processing normal messages and **ExpiredMLContainer** for processing near expired messages
- **ExpiredMessagesTimer** for refreshing the **ExpiredMLContainer** container filter

Some **BusinessInterface**, which is free from any transport-related information, interfaces, annotations, dependencies on infrastructure classes, and so on.

- The only requirement for this interface is that all the classes it uses are serializable or externalizable in some standard way.
- The technical agnostic nature of **BusinessInterface** allows us to use different types of transport in testing and in various deployment schemes and consequently different styles of realizing synchronous communication through the available infrastructure and environments.
- Through format conventions for methods, we can implement appropriate calls at transport level as either synchronous or asynchronous.

Transport configuration with properties

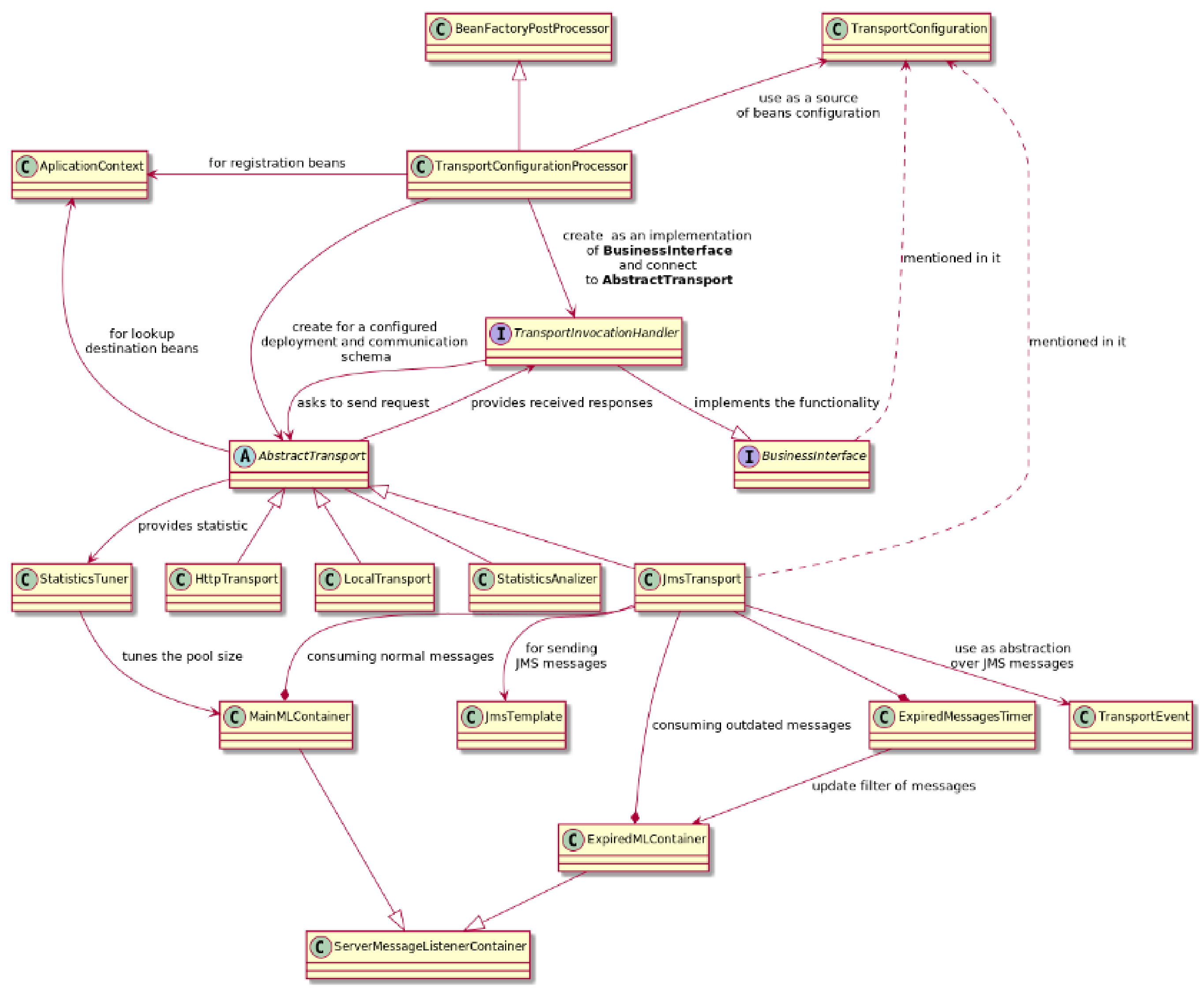
- **transportType** — HTTP / JMS / Local
- the list of business interfaces, which will be implemented as the remote proxies, working through this transport
- properties, specific for some transport. For example, for **JmsTransport** — input/output queue names, persistence, transaction style, the size of the thread pool, and so on…
- the list of beans, methods of which are exported for the remote access through the transport

Magic **TransportConfigurationProcessor** which extends **BeanFactoryPostProcessor** and

- for each transport in the configuration creates the corresponding spring-based service as an concrete implementation of **Transport**

- for each business interface mentioned in the configuration creates the corresponding spring-based service connected to the transport service.
- The remote proxy bean can be implemented by methods **Proxy.newProxyInstance()** and **ConfigurableListableBeanFactory.registerSingleton()**.
- Also, an instance of **TransportInvocationHandler** has to be implemented for packaging data from arguments of **BusinessInterface** into **TransportEvent** and back.

In general, this set of components can be illustrated by the following diagram:


BeanFactoryPostProcessor
TransportConfiguration
use as a source of beans configuration
AplicationContext
for registration beans
TransportConfigurationProcessor
create as an implementation of BusinessInterface and connect to AbstractTransport
mentioned in it
for lookup destination beans
create for a configured deployment and communication schema
TransportInvocationHandler
mentioned in it
asks to send request
provides received responses
implements the functionality
AbstractTransport
BusinessInterface
provides statistic
StatisticsTuner
HttpTransport
LocalTransport
StatisticsAnalizer
JmsTransport
tunes the pool size
consuming normal messages
for sending JMS messages
use as abstraction over JMS messages
MainMLContainer
JmsTemplate
consuming outdated messages
ExpiredMessagesTimer
TransportEvent
update filter of messages
ExpiredMLContainer
ServerMessageListenerContainer

This design also provides us with the following benefits:

- The problem of gathering and analyzing runtime statistics can be partially mitigated by the fact that both sides of the communication (remote proxy and server-side code) are represented as Spring beans.

a) Tools like “JavaMelody” allow us to collect various statistics at the boundaries of beans. Such an operation causes almost no extra workload and can be done in a production environment.

b) Accumulated statistics can be viewed both by accessing the web UI of the service itself (or by download the PDF with snapshot report) or by exporting for centralized storage and analysis through the “JavaMelody Prometheus Exporter”

- This transport layer is exceptionally compact, transparent, and integrates well with the rest of the stack technologies. For example, any transport can be used with the popular Resilience4j library, whose annotations can be applied on both sides of the corresponding interfaces.
- In this case, client and server are equal, which means we can make calls from both sides.

A spoonful of Kafka in a barrel of honey

As I already mentioned in my article **“Some detailed differences between Kafka and ActiveMQ”**, the main differences between Kafka and traditional MQ brokers are the following:

- the degree of parallelism of the consumption process is limited by the number of partitions
- the broker itself is “ dumb” and, in particular, does not provide the option to subscribe to a filtered message stream

On the one hand, there seems to be nothing to prevent the techniques described above from being reproduced based on Kafka. The role of unique client and server identifiers, in this case, will be partition numbers. If you dig around, you can even find a Spring component, **ReplyingKafkaTemplate**, which simplifies such implementation.

Alas, if we consider the problem more closely, it becomes apparent that the usage Kafka in this context is near useless.

- The maximum possible number of server and client instances is limited at the top by the number of partitions. Therefore, it is only possible to scale up the performance of the cluster to a limited extent.

a) Of course, it is possible to foresee a sufficiently large number of them in advance, but keeping each partition running means memory consumption for metadata storage and many different threads started in consumers and brokers, which consume memory and CPU resources.

b) The monitoring system tools required to operate distributed applications also take additional resources and sometimes very significant ones. For example, I have recently faced an incredible bug when initializing Kafka Binder for reading N topics created (1+N)*N/2 auxiliary threads. These threads were supposed to be used by the monitoring subsystem, but they were just idle and consumed resources during the whole lifetime of the application instance. You can read more about the problem and solution here — https://github.com/spring-cloud/spring-cloud-stream-binder-kafka/issues/1081.

- If we use manual partition assignment to clients and servers, we have no automatic scalability or fault tolerance. Suppose we apply groups and automatic distribution of partitions among consumers. In that case, any repartitioning initiated by any client means that some amount of the responses from already processed messages potentially will be delivered to other client instances who don't know what to do with them. In both cases, the construction is highly fragile and unstable to any problems.
- Due to each server's exclusive ownership of a set of partitions, all messages from each client will be mostly (until the next repartitioning occurs) handled by one server. Thus, we do not need to implement "weak sessions specifically." But "stealing

messages” close to expiration won’t work too — reading partition messages is possible only sequentially.

Leapfrogging of messages

Another potential problem is that most MQ brokers do not keep the order of messages sent around the same time. Moreover, even if they did save it, it would be rather pointless, as the receiving and processing of messages are done competitively by a set of threads from the container pool. Thus, the processing time of each particular message is in no way correlated with the processing times of its close neighbors.

In most cases, this is not a problem, as the different messages relate to different data aggregates, and the mutual order of handling is not very important. However, several scenarios are critical: for instance, the transmission of large fragments of data broken up into many individual chunks. Some brokers, ActiveMQ, for example, support sending and receiving of huge messages natively. For others, such as IBM WebSphere MQ, we need to perform it manually.

In this case, we have to consider that these messages can be received not only in a different order, but also (what is quite substantial) by different service instances. So, the intermediate storage of partial results of already processed messages in memory is therefore not possible. Yes, performing multiple queries on a relational database means some delay. But in the enterprise world, we can very rarely do anything without accessing the database. So this approach is reasonable in terms of the performance and throughput of the system as a whole.

The implementation of this functionality may be performed based on the following steps:

- Each message contains in its headers: the unique sequence ID as a **SeqId** header and the count of messages in the sequence as a **SeqLength** header
- In the database there are two tables: **T_SEQUENCES** with columns **SEQ_ID**, **SEQ_LENGTH** and **T_CHUNKS** with columns **MESSAGE_ID**, **SEQ_ID**, and **DATA**

When we receive any message

- first of all, check if there is a corresponding entry **<SeqId,SeqLength>** in the **T_SEQUENCES** table. If not, try to insert in a separate transaction, ignoring the possible uniqueness error of the primary key.
- obtain exclusive access to the sequence by starting the transaction and running **SELECT FOR UPDATE FROM T_SEQUENCES WHERE SEQ_ID = <SeqId>** query
- If all previous chunks have already been received: combine the data, process it and delete the corresponding records from both tables
- in other cases: insert the next fragment into **T_CHUNKS** table
- commit transaction

This simple pattern illustrates two fundamental principles of data processing through asynchronous message sequences (from the java multithreading point of view, this is nothing new, we use "select for update" instead of "synchronized"):

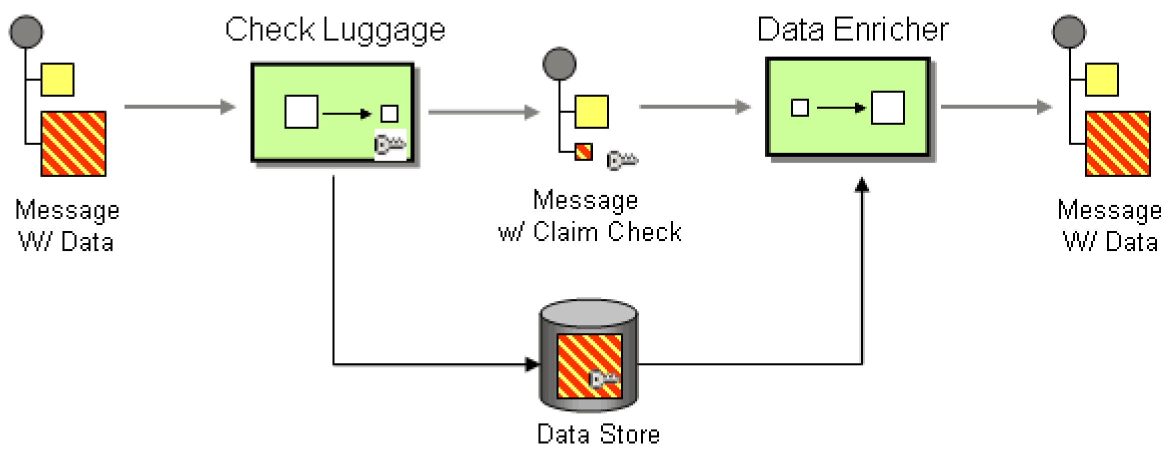
- all "if-then" constructions must be executed under protection of locks, which guards the data invariants
- The scope of the locking should be minimal to keep scalability

Some tips and improvements:

- If we deal with the synchronous interaction based on asynchronous ones, the calling thread has to block the sequence's last message. Accordingly, once we have finished assembling and processing a large amount of data, we use the ID of the last message as a **CorrelationID** header.
- If the system is heavily loaded, horizontal partitioning can reduce the degree of concurrency when accessing these tables. For example, in the most primitive case, we can define ten pairs of tables stored on different physical media and store data from each build process in tables **T_SEQUENCES_N** and **T_CHUNKS_N**, where N is the remainder of the sequence number divided by 10.

- To properly handle duplicate messages, it is recommended to implement logical deletion from the **T_SEQUENCES** table. In this way, if a message belongs to an already processed sequence and no corresponding entry remains in table **T_CHUNKS**, it can still be easily identified and ignored. A dedicated process should periodically clear the **T_SEQUENCES** table from already logically deleted entries older than a certain age.

Also, it is necessary to mention that the best way to solve such a puzzle is not to play with it. The set of enterprise integration patterns includes, in my opinion, the best solution for this task — the “Claim Check” pattern:



The only problem with this approach is that the data storage is usually non-transactional, so we have to implement also additional garbage collection mechanisms. Garbage in this context means “incorrectly added or not deleted files”.

A similar problem exists when we want to process an unordered stream of messages sequentially, applying each one individually. The treatment prescription is roughly similar, but we can meet with necessity to process each message from the received continuous sequence in a separate transaction. I’ll discuss this later in the section on message application patterns in the user interface.

By the way, in the case of Kafka, all the difficulties described in this section do not arise in principle.

Conclusion

So, currently, things are pretty good from a server perspective:

- The server thread usage time and the number of active threads are minimal.
- The database transaction time is also minimal, which ensures high database throughput
- The interaction is scalable for both sides and fault-tolerant
- The cohesion between the client and the server is significantly weakened comparing to the traditional approach.
- Both sides can operate in a transactional manner (but the whole process of performing a call and receiving a result cannot be completed in a single transaction, of course)
- We got a clear and easy-to-use monitoring indicator of the health for the both sides of the interaction — the length of the corresponding queue.

We could even go radically further and try to move entirely into a reactive paradigm, using NIO drivers to connect to the JMS and R2DBC to connect to the relational database. In the Spring technological stack case, there is an already available to use a combination of r2dbc-postgresql and Spring Data R2DBC. I have no personal experience of using such a combination, but it seems pretty possible to me.

However, we again encounter more or less the same problems, where a server operation again requires a synchronous call to a third service to get the result we need right now to complete the request.

The server runtime is now long blocked, database transaction are significantly increased, and overall performance suffers.

Suppose there are many synchronous calls in the system. In that case, the chain of dependencies between the individual services will spread the disease throughout the system, and it will become challenging to find a single point to apply the treatment.