Distributed Data Management
Akka Actor Programming Hands-On

Task 2 – LargeMessageProxy

Group: Robert'); DROP TABLE Students;-- (Joan Bruguera & Tom Braun)

Problem Statement

We wish to send a large message (a serializable object which will take at least a few MBs) between two Akka actors on different nodes.

This is not trivial, since messages between actors in different nodes are sent by Akka using a single channel. This can cause congestion for other (possibly more urgent) messages on the same channel. Therefore, a different side channel must be used.

The goal of this task is to design an actor, named LargeMessageProxy, that handles this large messaging logic between two sets of actors (a Master and several Workers) in a transparent way. Each Master and Worker actor has an associated child instance of the LargeMessageProxy actor on the same local actor system.

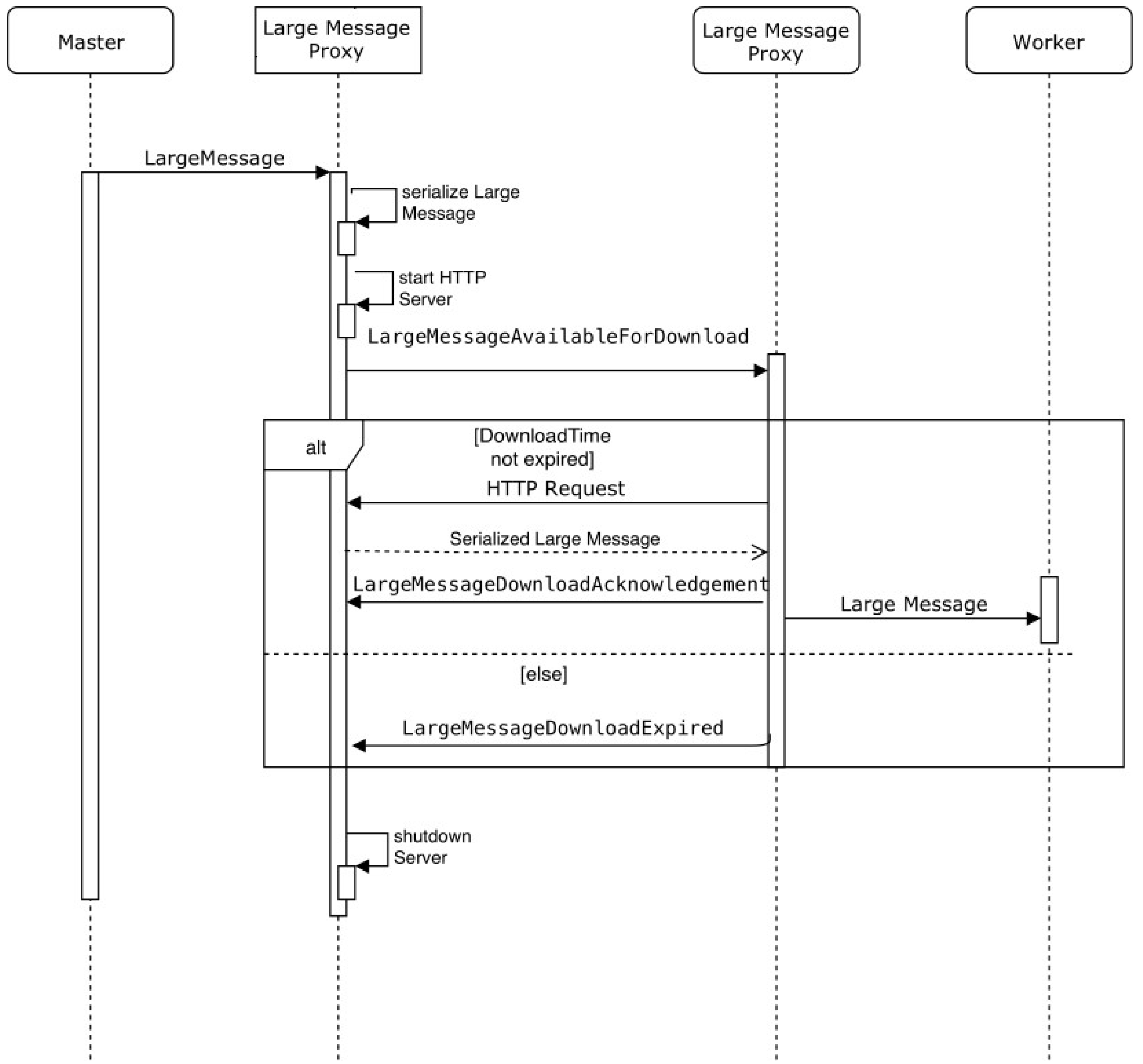
Our solution

Our LargeMessageProxy implementation does intra-node messaging but handles **serialization and message transfer** independently from Akka's message channel.

Basic schema:

- The sender (Master) sends the large message to the sender proxy.
- The sender proxy serializes the large message into a byte array.
- The sender proxy hosts the serialized content on an HTTP server.
- A message containing the URL is sent from the sender proxy to the receiver proxy.
- The receiver proxy downloads the file from the HTTP server.
- The receiver proxy deserializes the message into a Java object.
- The receiver proxy forwards the object to the receiver (Worker).

Our solution



Stack choice: Serialization

There are many choices available for object serialization. **We used Kryo for serialization** because:

- Compared to java.io.Serialization, it is widely reported to be more performant and similarly easy to use.
- Compared to Protocol Buffers / Thrift, it does not require additional complexity (definition files such as .proto / .thrift) which are not necessary in our simple case.
- Compared to XML / JSON, it uses binary serialization which results on a smaller amount of data to transfer.

Stack choice: Message transfer

Similarly, there are many choices available for transferring the serialized message. **We used the HTTP protocol through the Akka HTTP library** because:

- We found it relatively easy to use and integrate into our project.
- It takes care of all the low level details we may otherwise run into (chunking, routing, concurrency, etc.)

To transfer the files, we host them on a random route on the HTTP server, so they are publicly accessible through a URL like:

- http://123.123.123.123:12345/936da01f-9abd-4d9d-80c7-02af85c822a8

Afterwards, **a message containing this URL is sent to the receiver proxy**, which then downloads the serialized message (also using Akka HTTP in a non-blocking way).

Additional considerations

Resource release / avoiding memory leaks

We must eventually release the resources associated with the serialized message on the sender proxy when they are no longer necessary (i.e. when the receiver proxy has finished the download process).

- Otherwise, the sender proxy would eventually exhaust the available memory.
- To handled this, **the receiver proxy sends an acknowledgement message to the sender proxy** after the transfer is done. When this acknowledgement is received, the resources are released.

Additional considerations

Resource release / avoiding memory leaks

Even with the previous consideration, **a memory leak is still possible if messages are lost or nodes misbehave**.

- In this case, the acknowledgement message may never be received.
- We have aditionally **implemented a reasonable timeout (30s) for the receiver proxy to download the message**. If an acknowledgement is not received in this timeframe, the resources associated to the message on the sender proxy are also released
- (The message may be lost and a retry is not attempted)

Additional considerations

Throttling

- The Master actor sends a large message to every Worker actor when the connection is established.
- **If many Workers are simultaneously spawned, an out of memory situation is easily possible.**

Dirty solution: This can be easily fixed by a serializer cache, since the Master actor always sends the same class instance, which is repeatedly serialized

- However, we consider depending on this an implementation detail

Instead, **our solution is to throttle the amount of large messages simultaneously being transfered**:

- We transfer up to 20 messages simultaneously and the rest are (non-blockingly) stored into a queue and handled when possible (acknowledgement or timeout of another message).

Challenges and limitations

Challenges

- Akka is a very large and complex library so has a noticeable barrier to entry.
- Akka moves fast so it's easy to find documentation or examples that are outdated or incomplete.
- We ran into obstacles with alternative implementation choices (Akka Streams) which may be due to inexperience or due to real limitations.

Limitations

- We could only test our implementation on small setups (2 machines or a few containers) & not on a real distributed system.
- Due to time limitations we were unable to benchmark and evaluate our solution compared to other reasonable approaches.
- Better solutions may be possible in Akka (reinventing the weel)