Distributed Algorithms - Lab setup - 2020

RabbitMQ based distributed messaging

Authors

David Zwart (4224582)

IN4150 - Distributed Algorithms course

Embedded Systems
Technical University of Delft
Delft

26th of January. 2020

1 IN4150 previous and proposed lab

The previous lab of the course IN4150 is based around developing algorithms in Java Virtual Machine. The lab makes students implement an algorithm using Java RMI, which allows invoking methods on components bound to a registry. This RMI approach has the benefit of isolating code in a synchronous manner. The major disadvantage is how complex the RMI setup can be and how sensitive to system setup (firewall, IP and port configuration). Also, the RMI registry determined how the required classes had to be mapped to a Stub interface, something was not clear and would never be used in a real embedded-system. This document aims to propose a new lab structure to make the students familiar with at least one messaging system which is actively used these days and thus spark their interest in others after that.

The proposed lab focuses on using a modern messaging protocol, relieves the student of having to use Java and gives the ability to make messaging more visual by introducing a messaging broker. This broker is seen as an intermediary which fits much nicer with the course's way of explaining algorithms: messages, events and queue's. Compared to RMI, this queue based approach does not hide the atomic events of message transmit and arrival. This is a major gain, because the course seems to want to show how distributed messaging/events works in distributed algorithm, while RMI hides many interesting features of this.

Two major benefits of queue's over RMI:

- We cannot see the content of a RMI channel, while with queue's we can.
- Queue's can have unicast/multicast and broadcast options, RMI only unicast

1.1 RabbitMQ description & installation

The RabbitMQ messaging broker is a server compatible with almost all major coding languages through widely available libraries (incl Java, C#, NodeJs and Python). The primary target of this so-called **messaging broker** is to add persistence and fallback to AMQP messaging. AMQP is a duplex messaging protocol with delivery-guarantee among it's primary properties. RabbitMQ implements the 0-9-1 version of AMQP, meaning they implement message guarantees as well as the protocol.

The broker has very nice plugins like the **built-in management plugin** as shown in Figure 2, which extends the messaging server with a dashboard. This dashboard visualizes the state of the server (like channels, queue's and waiting messages). Finally, the server is provided primarily in an executable setup as well as many different possible docker images (f.e. *rabbitmq:3.8.2-management*). An executable installation is not system-isolated, where a docker-container is, which is better for reprodicibility.

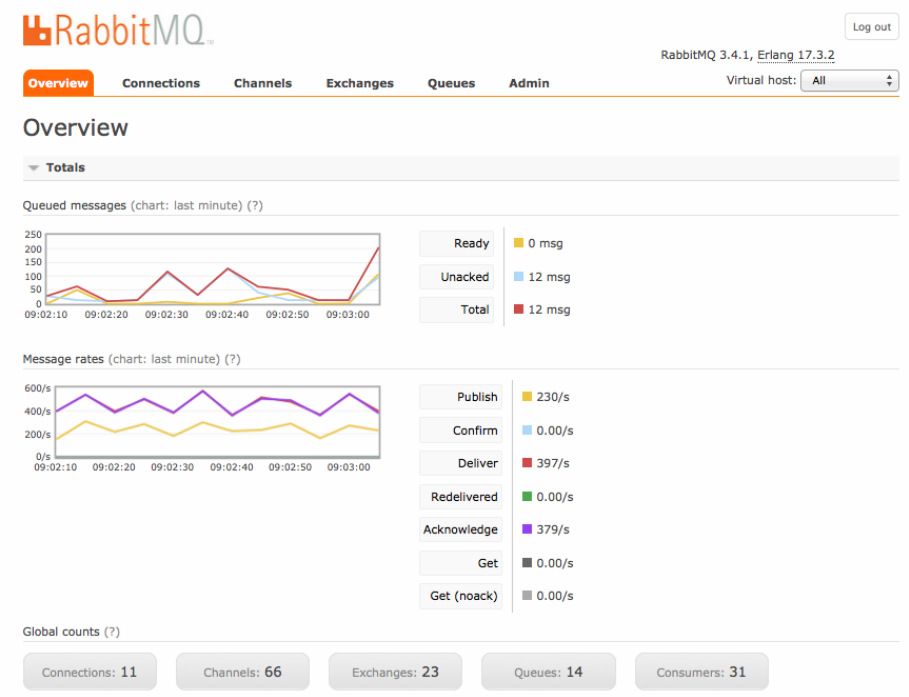

Figure 1: RabbitMQ management dashboard (plugin)

Students using Windows 10 Home are required to install the setup version of RabbitMQ-server, because docker won't run on Windows Home solely. The setup and plugin installation is easy and should not pose problems for Linux, Mac (homebrew) or Windows.

2 Example code for Chandy-Lamport's global state algorithm

First, we mention the library choice and explain our choice as to why/why not. Secondly, we explain the algorithm implementation and our view on what students should receive to work with from the beginning.

2.1 Python RabbitMQ with *aio_pika*

The example project is setup in python, because python is very easy to learn. I thought it would be better to provide an example for a language in which I hadn't worked with RabbitMQ yet. Since it still was a success and I've worked with C# and RabbitMQ before succesfully as well, I believe Java will also be very well doable, making this RabbitMQ approach a justified foundation for the lab. One major lesson however: asynchrony is a major benefit for the performance of the experiments. More on that below.

Initially the library *pika* was used following the RabbitMQ-python tutorial. This library **isn't asynchronous**, which is a loss: co-routine asynchrony is supported in python these days by using the *asyncio* library. After a small struggle with *pika*, the library *aio_pika* (independent of *pika*) is used with success to implement the asynchronous version of Chandy-Lamport's global-state algorithm. The details were found in the lecture notes of the course IN4150.

2.2 Code structure

The code is executed from **main.py**, which runs many **AlgorithmNode** instances within a thread pool. The algorithm mainly revolves around the classes **BaseNode**, **AlgorithmNode** and **AlgorithConfig**.

When the main function is ran, it creates a **ThreadPool** instance which triggers a synchronous function *start_async_node*() in the main file from within a new thread. This function spawns an AlgorithmNode instance, which extends from BaseNode. The AlgorithmNode class should implement the abstract functions **setup_connection()**, **run_core()** and a **receive_message()**) callback. The **setup_connection()** function allows the student to configure the type of message setup: fanout (broadcast), topic (multicast) and direct (unicast). It requires a bit of understanding as to how these types of messaging work, to which we refer the student to the RabbitMQ documentation as they have very good explaination on the different types.
Secondly, the async function **run_core()** in AlgorithmNode is called from the parent class BaseNode, once the connection to the RabbitMQ is succesful. This is where the student can do whatever they want (f.e. sending random messages (our choice), starting any algorithm or sleeping/awaking).
Finally, and most importantly, we send a kickstart message by calling *kickoff_simulation*() from within the main file to cause one node to 'spontaneously' start our specific algorithm. This message is sent by fanout (broadcast). Every node sends an acknowledge, as this is what RabbitMQ requires on a message level. After that the **receive_message()** function is called, where each node handles incoming messages in different ways. One important detail there is that the type of a message is used to decide on what to do with it.

2.2.1 Message types

For this specific algorithm we have different message types:

- PRE_INITIATION
- INITIATION
- MARKER
- GENERIC_TRANSFER

Generic transfer is meant for any traffic not related to the global-state algorithm, all others are for the distributed algorithm being simulated. I've chosen to make the generic traffic resemble bank transactions (random amounts of money being sent over input/output channels). This is happening randomly during the global-state algorithm to show that we actually have messages in transit during recording of local state.

2.2.2 Diagram of implementation

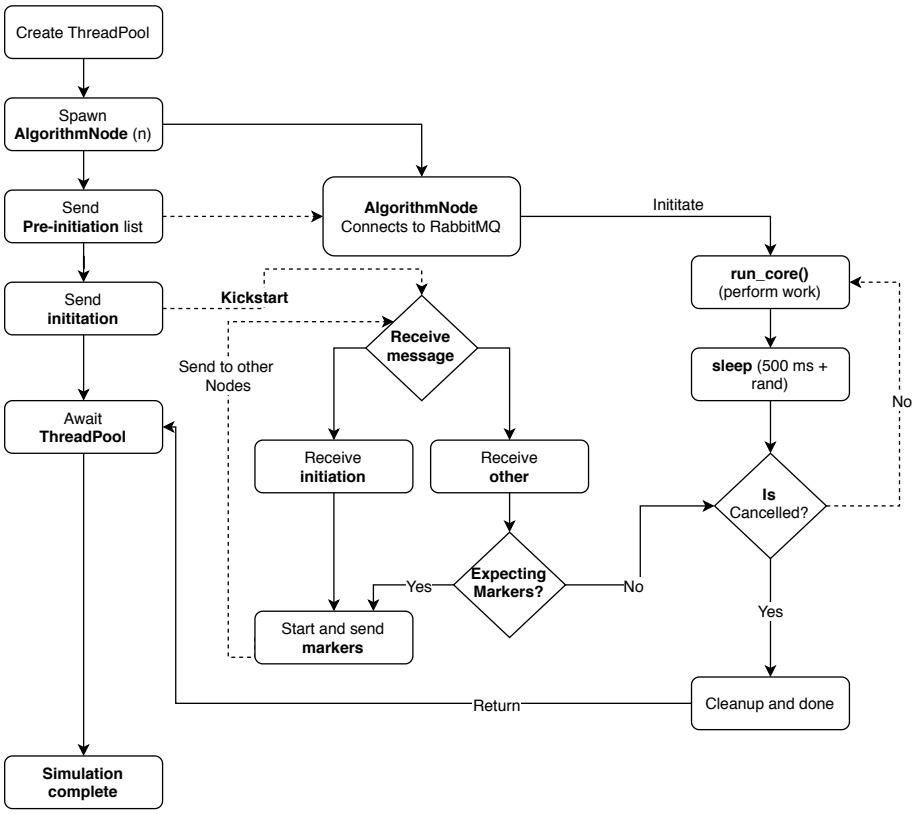

Figure 2: Simulation with asynchronous message-driven model. (Dashed line: message transfer, solid line: function call/return)

3 Caveats

Async vs sync messaging Avoid synchronous single-routine work as much as possible, as the core threading code would either waste CPU time on waiting for events as well as code not running as fast as possible. Use **wait** events to let **asyncio** handle any idling.

Asyncio eventloop Students have to get familiar with the event-loop system of *asyncio* in order to fully comprehend that the system is doing coroutine work for them: a single event-loop is checking arrival of many events with coroutines. This is not interesting for a student, until it goes wrong. For example, when an asynchronous function is not awaited, asyncio will print this in the terminal clearly, allowing the student to fix it.

However, creating an event-loop or trying to get a reference to a running event-loop is not always obvious as it is managed within the asyncio scheduler.

RabbitMQ setup As RabbitMQ has many features, it is possible that students don't understand the effect of different exchange types: fanout, direct or topic. They should get familiar with them, or the lab should provide a base template for the code. This way we avoid that students stray off to difficult implementations.

In our case, I expect the 'topic' exchange type to be of limited usefulness. We rarely need to explicitly filter messages to different receivers in a complex way, so my advice is to just use fanout for broadcast/multicast and direct for unicast.