Technical University of Moldova
Faculty of Computers, Informatics and Microelectronics
Department of Computer Science

REPORT

Laboratory no. 1-2

Message Broker

Author:
Andrei Capastru

Lecturer:
Dumitru Ciorbă

January 20, 2016

Contents

List of Figures

1 Theoretical background [1]

1.1 What is a message broker?

Message Broker is an intermediary program module which translates a message from the formal messaging protocol of the sender to the formal messaging protocol of the receiver. Message brokers are elements in telecommunication networks where programs (software applications) communicate by exchanging formally-defined messages.

The message itself is simply some sort of data structure such as a string, a byte array, a record, or an object. It can be interpreted simply as data, as the description of a command to be invoked on the receiver, or as the description of an event that occurred in the sender.

1.2 About brokering

In a distributed environment one usually deals with a large number of message senders and receivers which are part of the same system.

The message passing architecture is a very interesting and useful concept. The distinctive idea one can extract from it is the enabling of the loose coupling of the applications, in other words, the decoupling of the producers and consumers of messages, both in time and in space. As, a consequence, a lot of options have appeared for improving the performance of the system. For instance, the producer and consumer applications can run on different machines and at different times.

One illustrative example of the messaging architecture is the SMS communication architecture, in which the producer of the message sends the message to the consumer (via an intermediary entity which redirects the message to the consumer) and the latter receives the message, at a later point in time. The key aspect in this example is that the potential consumer needs not to be connected to the network at the moment of message sending.

1.3 The broker

A message broker is a physical component that handles the communication between applications. Instead of communicating with each other, applications communicate only with the message broker. An application sends a message to the message broker, providing the logical name of the receivers. The message broker looks up applications registered under the logical name and then passes the message to them.

Communication between applications involves only the sender, the message broker, and the designated receivers. The message broker does not send the message to any other applications. From a control-flow perspective, the configuration is symmetrical because the message broker does not restrict the applications that can initiate calls. Figure 1 illustrates this configuration.

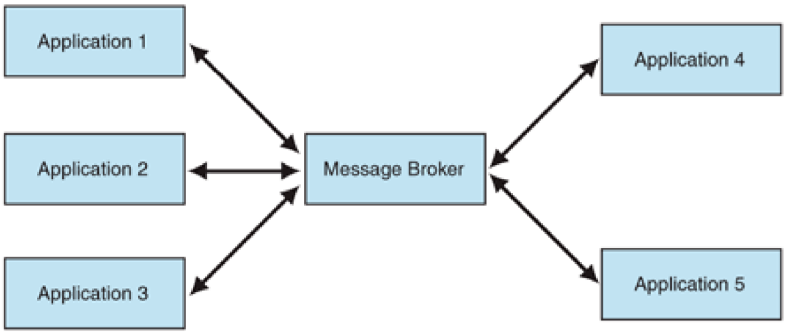

Figure 1: Message Broker

2 Implementation technologies

2.1 Peer-to-peer

In a peer-to-peer architecture, every node is directly responsible for the delivery of the message to the receiver. This implies that the nodes have to know the address and port of the receiver and they have to agree on a protocol and message format. The broker eliminates these complexities from the equation: each node can be totally independent and can communicate with an undefined number of peers without directly knowing their details.

A broker can also act as a bridge between the different communication protocols, for example, the popular RabbitMQ broker (http://www. rabbitmq.com) supports Advanced Message Queuing Protocol (AMQP), Message Queue Telemetry Transport (MQTT), and Simple/Streaming Text Orientated Messaging Protocol (STOMP), enabling multiple applications supporting different messaging protocols to interact.

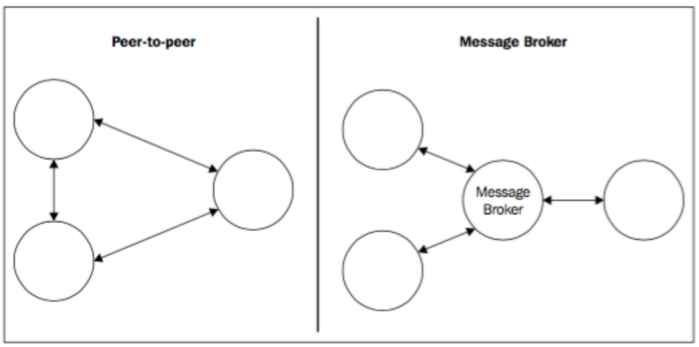

Figure 2: Peer to Peer vs. Message Broker Technologies

[1] **Benefits:**

- Reduced coupling. The message broker decouples the senders and the receivers. Senders communicate only with the message broker, and the potential grouping of many receivers under a logical name is transparent to them.

- Improved integrability. The applications that communicate with the message broker do not need to have the same interface. Unlike integration through a bus, the message broker can handle interface-level differences. In addition, the message broker can also act as a bridge between applications that are from different security realms and that have different QoS levels.

- Improved modifiability. The message broker shields the components of the integration solution from changes in individual applications. It also enables the integration solution to change its configuration dynamically.

- Improved security. Communication between applications involves only the sender, the broker, and the receivers. Other applications do not receive the messages that these three exchange. Unlike bus-based integration, applications communicate directly in a manner that protects the information without the use of encryption.

- Improved testability. The message broker provides a single point for mocking. Mocking facilitates the testing of individual applications as well as of the interaction between them.

[1] **Liabilities:**

- Increased complexity. Communicating through a message broker is more complex than direct communication for the following reasons:
  - The message broker must communicate with all the parties involved. This could mean providing many interfaces and supporting many protocols.
  - The message broker is likely to be multithreaded, which makes it hard to trace problems.

- Increased maintenance effort. Broker-based integration requires that the integration solution register the applications with the broker. Bus-based integration does not have this requirement.

- Reduced availability. A single component that mediates communication between applications is a single point of failure. A secondary message broker could solve this problem. However, a secondary message broker adds the issues that are associated with synchronising the states between the primary message broker and the secondary message broker.

- Reduced performance. The message broker adds an intermediate hop and incurs overhead. This overhead may eliminate a message broker as a feasible option for solutions where fast message exchange is critical.

2.2 Asynchronous messaging and queues

An important advantage of asynchronous communications is that the messages can be stored and then delivered as soon as possible or at a later time. This might be useful when the receiver is too busy to handle new messages or when we want to guarantee the delivery. In messaging systems, this is made possible using a message queue, a component that mediates the communication between

the sender and the receiver, storing any message before it gets delivered to its destination, as shown in the following figure:

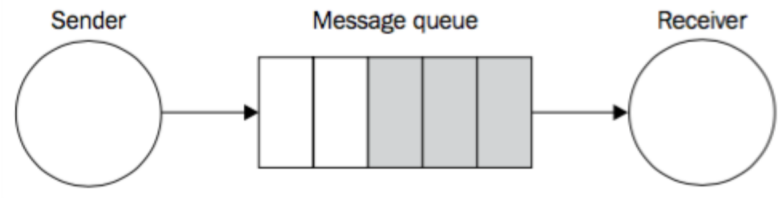

Figure 3: Message Queue

2.3 Pub/Sub Pattern

Publish/Subscribe [2] (often abbreviated Pub/Sub) is probably the best known one- way messaging pattern. We should already be familiar with it, as its nothing more than a distributed observer pattern. As in the case of observer, we have a set of subscribers registering their interest in receiving a specific category of messages. On the other side, the publisher produces messages that are distributed across all the relevant subscribers.

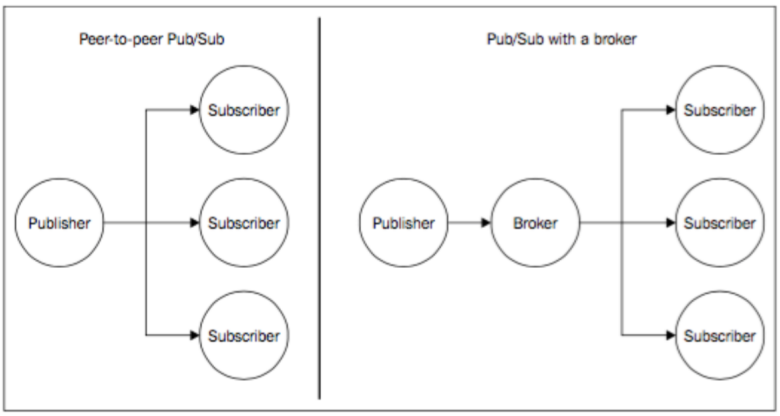

Figure 4: P2P Pub/Sub vs. Pub/Sub with a broker

The above figure shows the two main variations of the pub/sub pattern, the first P2P, the second using a broker to mediate the communication.

A big advantage of Pub/Sub pattern is the fact that the publisher doesnt know who the recipients of the messages are in advance.

The presence of a broker further improves the decoupling between the nodes of the system because the subscribers interact only with the broker, not knowing which node is the publisher of a message.

3 The Application

3.1 Stack description

The application is based on Node.js asynchronous event driven framework. Node.js was designed to build scalable network applications and scalability.

Thread-based networking is relatively inefficient and very difficult to use. Furthermore, users of Node are free from worries of deadlocking the process there are no locks. Almost no function in Node directly performs I/O, so the process never blocks. Because nothing blocks, less-than- expert programmers are able to develop scalable systems.

Node is similar in design to and influenced by systems like Rubys Event Machine or Pythons Twisted. Node takes the event model a bit further, it presents the event loop as a language construct instead of as a library. In other systems there is always a blocking call to start the event-loop.

Typically one defines behavior through callbacks at the beginning of a script and at the end starts a server through a blocking call like `EventMachine::Run()`.

In Node there is no such start-the-event-loop call. Node simply enters the event loop after executingNode.js the input script. Node exits the event loop when there are no more callbacks to perform. This behavior is like browser JavaScript the event loop is hidden from the user.

Libraries used:

- `fs` — FileSystem Library [3]
- `dgram` — UDP Datagram Library [4]

3.2 Solution description

3.2.1 Source code

The source code is divided into the files:

- `broker.js` — the code of messaging broker.
- `sender.js` — the code of sending entity.
- `receiver.js` — the code of receiving entity.
- `sent.xml` — xml file that contains the info to be sent.

3.2.2 Work flow

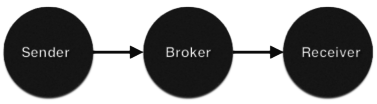

Figure 5: Work Flow Diagram

The following steps are done during running the application:

1. Sender reads sent.xml using `fs::readFile` function. [3]
2. Sender sends the content to broker using `dgram::send` function. [4]
3. Broker forwards the content to receiver using `dgram::send` function. [4]
4. Receiver writes the message to received.xml in xml format using `fs::writeFile` function. [3]

4 Conclusion

During this laboratory work I got to know one of the most used patterns in distributed applications: Message Broker Pattern and its variations.

I also got familiarized with Node.js (after long decision making process between Ruby, Java, Python, C# and Node.js). I decided to use it because I never did it before, all other I used in various activities (mostly Ruby).

I never used JavaScript as a server-side language before and Node.js allowed me to do it. Was it a wise decision? Perhaps not. Did I learn experienced something new? Definitely yes. As JavaScript is increasing in popularity and efficiency (Google and Apple working on asm.js), this might become a main component in web architecture in near or not so near future.

It uses an event loop to reduce the awkward (and inefficient) process of executing asynchronous I/O operations. In particular, it drops the memory requirements for this for large numbers of calls dramatically. The event loop is responsible for executing an asynchronous task and then passing back the result it also handles this by executing each task in the most efficient order.

That means developers working with Node.js can build a complex application that can easily handle the needs of millions of users. The event loop does the hard work and the application dealing with client requests doesn't have to.

But you should keep in mind the callbacks. Always.[5]

References

[1] MSDN. *Message Broker*. 2004. URL: https://msdn.microsoft.com/en-us/library/ff648849.aspx.

[2] MSDN. *Publish/Subscribe*. 2004. URL: https://msdn.microsoft.com/en-us/library/ff649664.aspx.

[3] Node.js. *FileSystem Node.js v5.4.1 Manual & Documentation*. 2012-2015. URL: https://nodejs.org/api/fs.html.

[4] Node.js. *UDP / Datagram Sockets Node.js v5.4.1 Manual & Documentation*. 2012-2015. URL: https://nodejs.org/api/dgram.html.

[5] TutorialsPoint. *Node.js Callbacks Concept*. 2015. URL: http://www.tutorialspoint.com/nodejs/nodejs_callbacks_concept.htm.