Technical University of Moldova
Faculty of Computers and Information Technologies
Specialty Computer Science

Report

Laboratory #1

Message Broker

Authors:
Cristian Cartofeanu

Lecturer:
Dumitru Ciorbă

October 30, 2015

Contents

1 Objective

Integration based on messaging agents that would allow for asynchronous communication between the distributed components of a system (as defined in 1.1):

1.1 Definitions

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.

2 Theoretical Background

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. He may even have some service responsible for the communication shut down, but, as soon as the consumer turns on the service, it will receive the intended message.

As an additional specification of this architecture, there is no restriction on the use of software/hardware platforms on the producers/consumers side whatsoever. That means, that as long as they understand and use the same messaging protocol they can rely on whichever platform they wish.

Another core feature of the message passing architecture is that one can easily add new pieces in the system that can guarantee they will not affect the other parts.

Exempli gratia, a system is described like this:

"[...]each command is implemented as a part to the overall GUI. If we want to add a new feature, or modify an existing one, we just write a new part and add

it to the system. When it gets called, it has no dependencies on the rest. It means we can extend our app very easily."

In another example, we have:

"[...] a message passing between nodes on our network - when something gets changed in one computer, a message is sent to all others so they can update themselves. We have a hundred different messages for different events, so we can extend the system by dropping a new service in that reacts to the appropriate messages."

3 Solution Description (using Redis)

3.1 About Brokering

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 symmetric because the message broker does not restrict the applications that can initiate calls. Figure 1 illustrates this configuration.

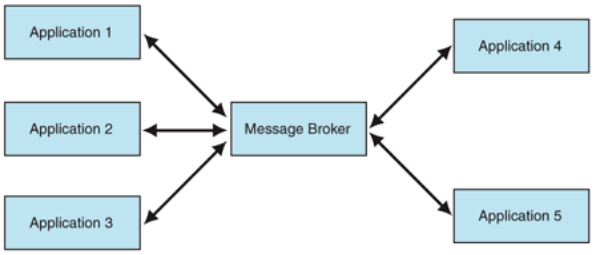

Figure 1: Message Broker

The message broker can expose different interfaces to the collaborating applications, and it can translate messages between these interfaces. In other words, the message broker does not enforce a common interface on the applications.

Table 1: Message Broker Responsibilities and Collaborations

Responsibilities	Collaborations
Receive message	Senders: applications that,send messages to the,message broker.
Determine the message recipients,and perform the routing	Receivers: applications that,receive messages from the,message broker
Handle any interface-level differences	
Send the message to the recipients	

Prior to using a message broker, you must register the applications that receive communications so that the message broker can dispatch requests to them. The message broker may provide its own registration mechanism, or it may rely on an external service such as a directory.

3.1.1 Why is it done for

Placing the message broker between the sender and the receiver provides flexibility in several ways. First, the message broker allows the integration solution to dynamically change its configuration. For example, if an application must be shut down for maintenance, the message broker could start routing requests to a failover application. Likewise, if the receiver cannot keep up with the incoming messages, the message broker could start load balancing between several receivers.

Second, the message broker can choose between applications that have different QoS levels. This resembles the dynamic configuration, but the message broker selects the application based on specified criteria. For example, an application for premium accounts may fulfil requests quickly, but an application for general use may have a longer processing time.

Third, the message broker allows the sender and the receiver to reside in different security realms. In other words, the message broker can reside on the boundary between two security realms and bridge requests between those two realms. Table 1 shows the responsibilities and collaborations of a message broker.

3.1.2 Benefits and Liabilities

The decision to use a message broker for integration entails balancing the benefits of removing inter-application coupling against the effort associated with using the message broker. Use the following benefits and liabilities to evaluate the balance:

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.

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.

3.2 Possible implementation technologies

3.2.1 Asynchronous messaging and queues

As Node.js developers, we should already know the advantages of executing asynchronous operations. For messaging and communications, it's the same story.

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:

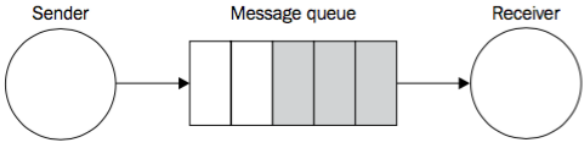

Figure 2: Message queue.

If for any reason the receiver crashes, disconnects from the network, or experiences a slowdown, the messages are accumulated in the queue and dispatched as soon as the receiver comes online and is fully working. The queue can be located in the sender, or split between the sender and receiver, or living in a dedicated external system acting as a middleware for the communication.

3.2.2 Peer-to-peer or broker-based messaging

Messages can be delivered directly to the receiver, in a peer-to-peer fashion or through a centralised intermediary system called Message Broker. In our example we use Message Broker. The main role of the broker is to decouple the receiver of the message from the sender. The following figure shows the architectural difference between the two approaches:

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 pop-

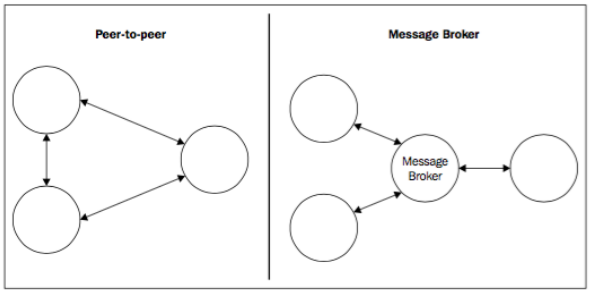

Figure 3: Peer to peer and message broker.

ular 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.

Besides the decoupling and the interoperability, a broker can offer more advanced features such as persistent queues, routing, message transformations, and monitoring, without mentioning the broad range of messaging patterns that many brokers support out of the box. Of course, nothing can stop us from implementing all these features using a peer-to-peer architecture, but unfortunately there is much more effort involved. Nonetheless, there might be different reasons to avoid a broker:

- Removing a single point of failure
- A broker has to be scaled, while in a peer-to-peer architecture we only need to scale the single nodes
- Exchanging messages without intermediaries can greatly reduce the latency of the transmission

3.2.3 Publish/subscribe pattern

Publish/subscribe (often abbreviated Pub/Sub) is probably the best known one-way messaging pattern. We should already be familiar with it, as it's 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. The following figure shows the two main variations of the pub/sub pattern, the first peer-to- peer, the second using a broker to mediate the communication:

What makes pub/sub so special is the fact that the publisher doesn't know who the recipients of the messages are in advance. As we said, it's the subscriber

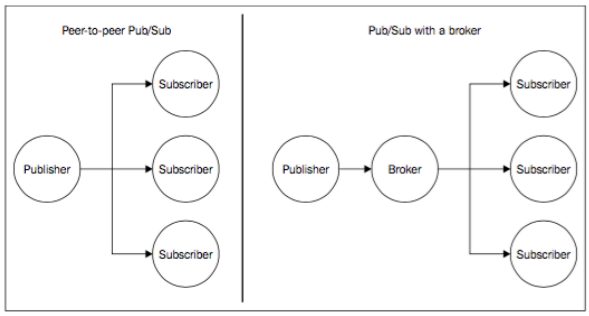

Figure 4: Publish/subscriber in peer to peer and message broker.

which has to register its interest to receive a particular message, allowing the publisher to work with an unknown number of receivers. In other words, the two sides of the pub/sub pattern are loosely coupled, which makes this an ideal pattern to integrate the nodes of an evolving distributed system.

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.

3.2.4 Libraries

a. Redis package
To connect our Node.js application to the Redis server, we use the redis package (https:// npmjs.org/package/redis), which is a complete client that supports all the available Redis commands. Next, we instantiate two different connections, one used to subscribe to a channel, the other to publish messages. This is necessary in Redis, because once a connection is put in subscriber mode only commands related to the subscription can be used. This means that we need a second connection for publishing messages.

b. WebSocketServer moudule
ws-is a simple to use WebSocket implementation, up-to-date against RFC-6455, and probably the fastest WebSocket library for node.js.
We use WebSocketServer in order to create a new instance and after we attach it to our existing HTTP server. We then start listening for incoming WebSocket connections, by attaching an event listener for the connection event.

3.3 Use cases (system's functionalities)

3.3.1 Actors

Our plan of action is to integrate our chat servers using Redis as a message broker. Each instance publishes any message received from its clients to the broker, and at the same time it subscribes for any message coming from other server instances. As we can see, each server in our architecture is both a subscriber and a publisher. The following figure shows a representation of the architecture that we obtain:

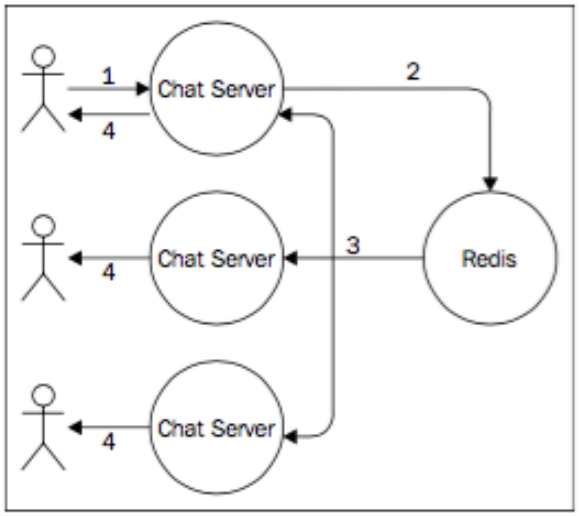

Figure 5: Use cases of our app.

3.3.2 Scenarios

By looking at the preceding figure, we can sum up the journey of a message as follows:

a. The message is typed into the textbox of the web page and sent to the connected instance of our chat server.

b. The message is then published to the broker.

c. The broker dispatches the message to all the subscribers, which in our architecture are all the instances of the chat server.

d. In each instance, the message is distributed to all the connected clients.

3.4 The applications project

3.4.1 Architectural description

From the architectural point of view, the application was designed using the Node.js asynchronous event driven framework along with the redis data structure server. Why Node.js? Because Node.js was designed to build scalable network applications and scalability, as a modern requirement, represents a key component in any distributed system.

This is in contrast to today's more common concurrency model where OS threads are employed. Thread-based networking is relatively inefficient and very difficult to use. Furthermore, users of Node are free from worries of deadlocking the processthere 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 Ruby's Event Machine or Python's 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 behaviour 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 executing the input script. Node exits the event loop when there are no more callbacks to perform. This behaviour is like browser JavaScript the event loop is hidden from the user.

HTTP is a first class citizen in Node, designed with streaming and low latency in mind. This makes Node well suited for the foundation of a web library or framework.

Just because Node is designed without threads, doesn't mean you cannot take advantage of multiple cores in your environment. You can spawn child processes that are easy to communicate with by using Node.js'`child_process.fork()` API. Built upon that same interface is the cluster module, which allows you to share sockets between processes to enable load balancing over your cores.

Redis is more a database than a message broker, however among its many features there is a pair of commands specifically designed to implement a centralized publish/subscribe pattern.

Of course, the implementation is very simple and basic, compared to more advanced message-oriented middleware, but this is one of the main reasons for its popularity. Often, in fact, Redis is already available in an existing infrastructure, for example, as a caching server or session store; its speed and flexibility

make it a very popular choice for sharing data in a distributed system. So, as soon as the need for a publish/subscribe broker arises in a project, the most simple and immediate choice is to reuse Redis itself, avoiding to install and maintain a dedicated message broker.

3.4.2 Design

The design of the app relies on the publish/subscribe pattern, which is probably the best known one-way messaging pattern. It is basically nothing more than a distributed observer pattern. As in the case of observer, there is 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. The following figure shows the main variation of the pub/sub pattern, which uses a broker to mediate the communication:

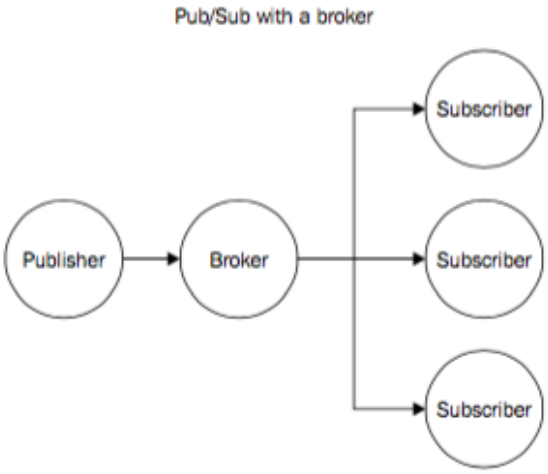

Figure 6: Publish/Subscribe with a broker

What makes pub/sub so special is the fact that the publisher doesn't know who the recipients of the messages are in advance. As we said, it's the subscriber which has to register its interest to receive a particular message, allowing the publisher to work with an unknown number of receivers. In other words, the two sides of the pub/sub pattern are loosely coupled, which makes this an ideal pattern to integrate the nodes of an evolving distributed system.

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. As we will see later, a broker can also providea message queuing system, allowing a reliable delivery even in the presence of connectivity problems between the nodes.

3.5 Source code

The source code is structured in three files:

- `index.html` : which is the file that is responsible for the html layout of our application. It is a frame for our sender(client) application, in which the sender introduces either the message manually, or attaches a file to be sent through the network to the chat server application.
- `function.js` : which isolates the business logic of the client application in a separate layer, not necessary available for a potential client.
- `app.js` : which is the frame, where the redis broker resides, along with the receiver(server)application

3.6 Functionality

Routing
To connect our Node.js application to the Redis server, we use the redis package (https://npmjs.org/package/redis), which is a complete client that supports all the available Redis commands. Next, we instantiate two different connections, one used to subscribe to a channel, the other to publish messages. This is necessary in Redis, because once a connection is put in subscriber mode only commands related to the subscription can be used.

This means that we need a second connection for publishing messages.

```
var redis = require("redis");
var redisSub = redis.createClient();
var redisPub = redis.createClient();
```

We create a new instance of the WebSocket server and we attach it to our existing HTTP server. We then start listening for incoming WebSocket connections, by attaching an event listener for the connection event.

```
var wss = new WebSocketServer({server:  server});
wss.on('connection', function(ws) {
console.log('Client connected');
}
```

Each time a new client connects to our server, we start listening for incoming messages. When a new message arrives, we broadcast it to all the connected

clients.

```
ws.on('message', function(msg) {
console.log('Message:  '  + msg);
```

When a new message is received from a connected client, we publish a message in the `chat_messages` channel. We dont directly broadcast the message to our clients because our server is subscribed to the same channel (as we will see in a moment), so it will come back to us through Redis. For the scope of this example, this is a simple and effective mechanism.

```
redisPub.publish('chat_messages', msg);
});
});
```

As we said, our server also has to subscribe to the `chat_messages` channel, so we register a listener to receive all the messages published into that channel (either by the current server or any other chat server). When a message is received, we simply broadcast it to all the clients connected to the current WebSocket server.

```
redisSub.subscribe('chat_messages');
redisSub.on('message', function(channel, msg) {
wss.clients.forEach(function(client) {
client.send(msg);
});
});
```

4 Solution description (without Redis)

4.1 Use cases (system's functionalities)

4.1.1 Actors

Our next plan of action is to integrate our chat servers using a message broker written by us. Each receiver publishes any message received from its broker who also received that message from a sender, and at the same time receiver translate the message and saves it in Json and xml format. As we can see, we have 1 sender 1 broker and 1 receiver. The following figure shows a representation of the architecture that we obtain:

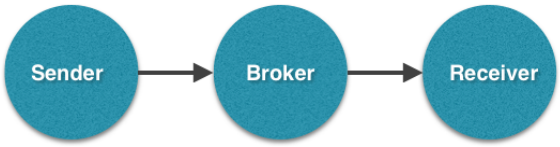

Figure 7: Use cases of our app.

4.1.2 Scenarios

By looking at the preceding figure, we can sum up the journey of a message as follows:

a. The message is send to it's broker.

b. The message is then published to the broker.

c. The broker dispatches the message to all the subscribers, which in our architecture are only one receiver.

d. Receiver translates the message in both Json and XML format and saves it.

4.2 Source code

The source code is structured in 4 files:

- `sender.js` : which is responsible for reading from the file and sending the file's data to the broker.
- `broker.js` : which choose a receiver for sending the message forward.
- `receiver.js` : which receives the message and translates it's content and saves it into a file.
- `index.xml` : File that contain the data we want to send.

5 Conclusions

We learned to work with one of the most important messaging and integration pattern (Message Broker) and the role it plays in the design of distributed systems.
We made our acquaintance with the one major type of message exchange pattern and namely, publish/subscribe, and we saw how it can be implemented using a message broker. We analyzed their pros and cons, and we saw that by using Redis as a message broker, we can implement reliable and scalable application

with little development effort but at a cost of lacking the whole functionality a full-fledged message broker would offer, such as ØMQ.

5.1 The advantages and disadvantages of the mechanisms we used

Node.js is the Solution?
Node.js is server-side JavaScript execution environment that is event-driven. It utilizes the V8 engine (used by Google in Chrome), which turns JavaScript into machine code. Thats much faster to execute than using an interpreter or running it as bytecode.

And, thats pretty handy since many developers already know JavaScript and thus Node.js doesn't require learning a language from scratch but is rather an extension of an easily available skill set.

Node.js has a secret weapon too. 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 doesnt have to.

The Downsides of Node.js

Node.js is new, and that means there are still some awkward moments for it. It's almost certainly going to be THE preferred language of web application development in the near future. However, right now you might want to think about:

- The Node API: There have been a few issues regarding API stability. These are getting more stable as releases progress but it's worth noting that not every API change has been compatible with previous releases.
- The Node.js Libraries: There's a shortage of standard libraries for JavaScript in general. This may improve as things move forward - or it may not - but that may mean more work for your application development team too.
- The newness of Node.js: Nobody knows where the pain points of working with large scale Node.js applications lie at the moment. You're going to

have to keep a careful eye on security, performance and maintenance issues for a while until the language has been more thoroughly road tested.

6 References

[1] Message Broker
[2] Hohpe G., Woolf B. Introduction to Messaging Channels [Online] // Enterprise integration patterns.
[3] Casciaro Mario, Node.js Design Patterns, 2014, Packt Publishing, ISBN-13: 978-1783287314
[4] Macedo T., Oliveira F., Redis Cookbook, 2011, O'Reilly, ISBN: 978-1-449-30504-8, p. 26
[5]https://github.com/CristianChris/Creating-Chat-Application-using-Message-Broker-technology-in-Node.js