Broker vs. Brokerless

ØMQ: The fastest messaging framework http://www.zeromq.org

Introduction

This article presents different models of how messaging can be done. It discusses drawbacks and advantages of individual approaches. It is meant as a background reading to learn how ØMQ differs from traditional messaging systems.

Broker

Architecture of most messaging systems is distinctive by the messaging server ("broker") in the middle. You can think of it as of classical "star" or "hub and spoke" architecture. Every application is connected to the central broker. No application is speaking directly to the other application. All the communication is passed through the broker.

There are several advantages to this model.

Firstly, applications don't have to have any idea about location of other applications. The only address they need is the network address of the broker. Broker then routes the messages to the right applications based on business criteria ("queue name", "routing key", "topic", "message properties" etc.) rather than on physical topology (IP addresses, host names).

Secondly, message sender and message receiver lifetimes don't have to overlap. Sender application can push messages to the broker and terminate. The messages will be available for the receiver application any time later.

Thirdly, broker model is to some extent resistant to the application failure. So, if the application is buggy and prone to failure, the messages that are already in the broker will be retained even if the application fails.

Drawbacks of broker model are twofold: Firstly, it requires excessive amount of network communication. Secondly, the fact that all the messages have to be passed through the broker can result in broker turning out to be the bottleneck of the whole system. Broker box can be utilised to 100% while other boxes are under-utilised, even idle almost all the time.

To demonstrate the drawbacks of the broker model, let's consider a simple scenario where data have to be processed by four distinct applications in a row. The pseudo-code for the scenario will look like this:

```
function AppA (x)
{
    y = do_business_logic_A (x);
    return AppB (y);
}

function AppB (x)
{
    y = do_business_logic_B (x);
    return AppC (y);
```

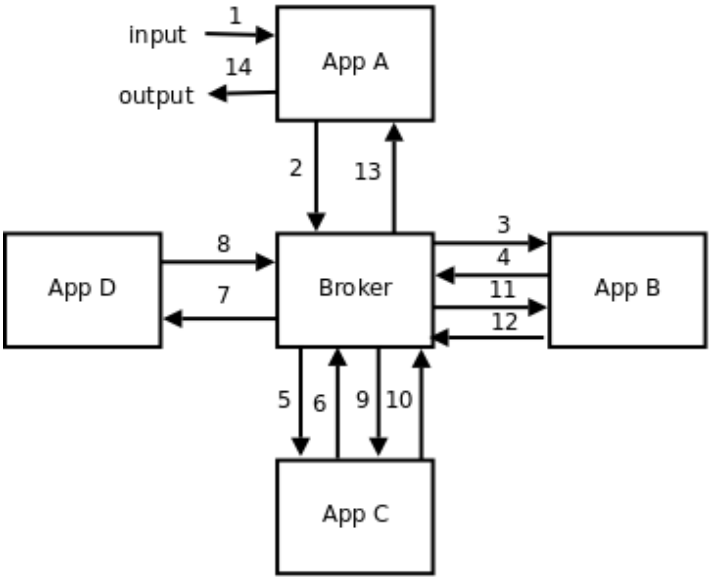
```
}

function AppC (x)
{
    y = do_business_logic_C (x);
    return AppD (y);
}

function AppD (x)
{
    return do_business_logic_D (x);
}
```

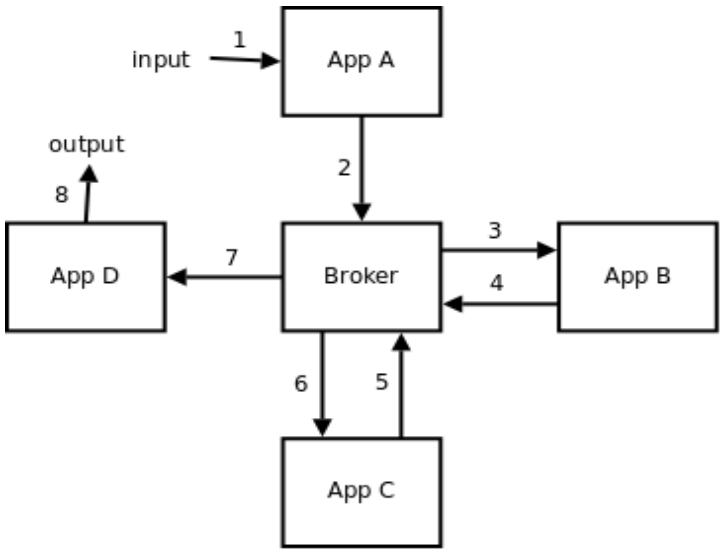
First, let's have a look how this scenario would look like when implemented using SOA (ESB, request/reply) architecture. This architecture is based on RPC (remote procedure call) paradigm, where one application "calls" a function in a remote application. This is done by packing (marshalling) arguments of the function and sending them across the network to the other application, where the arguments are unpacked (unmarshalled) and the function is processed. Result is then packed and sent back to the calling application, which unpacks it and continues with processing.

Using this paradigm we need 12 network hops to execute the scenario (input and output are not considered hops, they are simple local calls):



Even more importantly, broker has to process 6 messages (each message has to be passed in and out of the broker, thus 12 network hops) which is not much by itself, however, with high transaction rate (say 100,000 business transactions a second) the number of messages processed in the broker may hit the limit of the broker and/or hardware it is running on (600,000 messages a second).

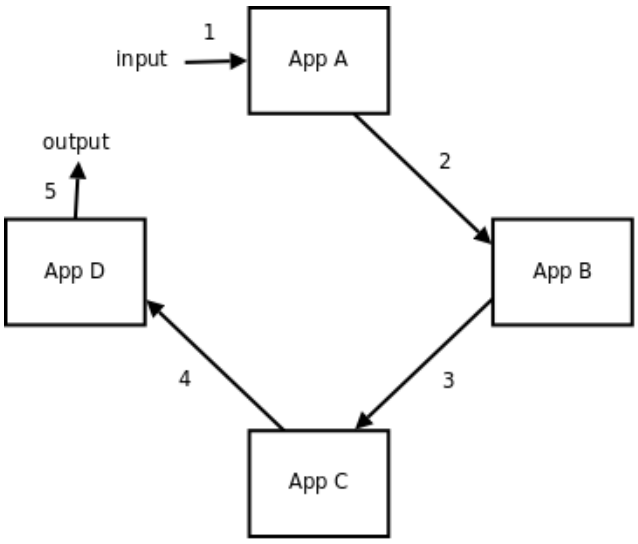
To lower the load on the broker and eliminate latency, we may choose to avoid SOA model and choose to implement the process in a pipelined fashion. That way we can avoid half of the messages (returning data from RPC "functions"). This kind of solution is depicted on the diagram below:



With a central broker architecture you cannot get more efficient than that. If the broker still acts like a bottleneck and/or latency is still too high, the only way to move forward is to eliminate the broker itself.

No Broker

Following diagram shows the scenario with applications sending messages each to another without the broker in the middle:



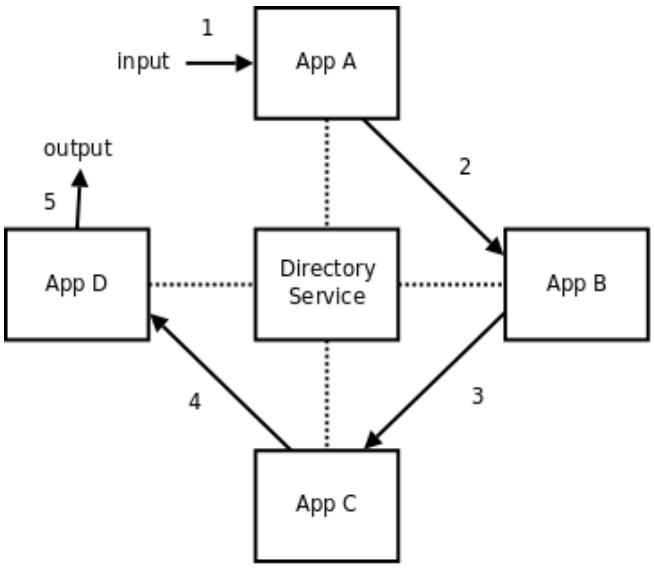
As can be seen number of network hops decreased to three and there is no single bottleneck on the network. This kind of arrangement is ideal for applications with a need for low latency and/or high transaction rate. The trade-off is worsened manageability of the system. Each application has to connect to the applications it communicates with and thus it has to know the network address of each such application. While this is acceptable in the case as simple as our example, in real world enterprise environment with hundreds of interconnected applications managing the solution would quickly become a nightmare.

Broker as a Directory Service

Note that we can split the functionality of the broker into two separate parts. Firstly, broker has a repository of applications running on the network. It knows that application X runs on host Y and that messages intended for X should be sent to Y. In acts like a directory service. Secondly, broker does the message transfer itself.

To solve the manageability issue we can leave the former functionality in the broker and shift the message transfer to be done by applications themselves. Thus, application X will register with the broker letting it know that it runs on box Y. Application Z wanting to send a message to application X will query the broker for the location of X. Once the broker replies that X is located on box Y, Z can create a connection directly to Y and send the message itself without bothering broker at all.

Following picture shows this architecture:



This way we can get high performance and manageability at the same time. To get more details, ØMQ exchange example is an implementation of "broker as a directory service" architecture.

However, in many cases there are few more problems to solve.

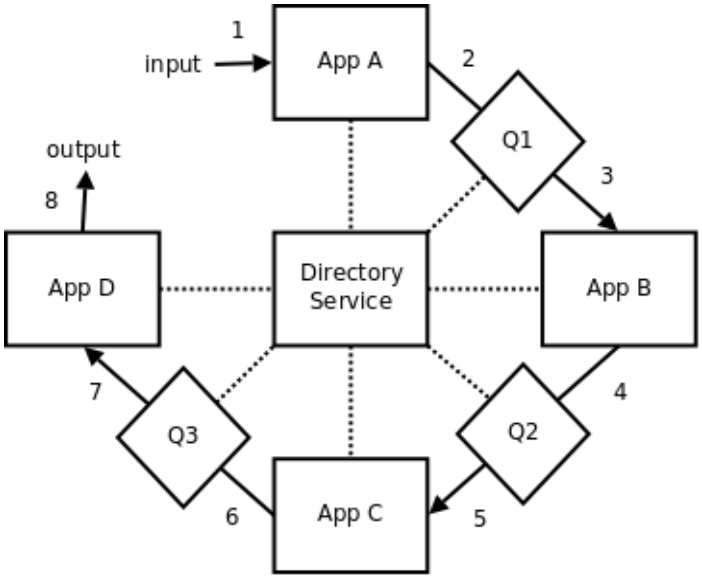
Distributed broker

As was already said, there are some advantages to the broker model, that are not available with the brokerless model.

The sender application and the receiver application don't have to have overlapped lifetimes. The messages are stored in the broker while sender is already off and receiver has not yet started. Also, if the application fails, the messages that were already passed to the broker are not lost.

To achieve this kind of behaviour you simply have to have some application (broker) in the middle. Consequently, you cannot avoid 2 network hops to get message from sender to the receiver, but still, it would be nice to avoid the "broker as a bottleneck" problem.

The "distributed broker" architecture does exactly that:



As shown on the diagram, each message queue is implemented as a separate application. It may run on the same box as one of the applications it is connecting, it may be located on a completely different box. Several queues may run on a single box, the box may be dedicated exclusively to host a single queue. Queue is registered with the broker (directory service) and thus it is accessible to all the applications on the network. Moreover, the queue is very simple piece of software that's getting messages from senders and distributing them to the receivers. so the chance of failure is much lower that with real applications full of complex business logic.

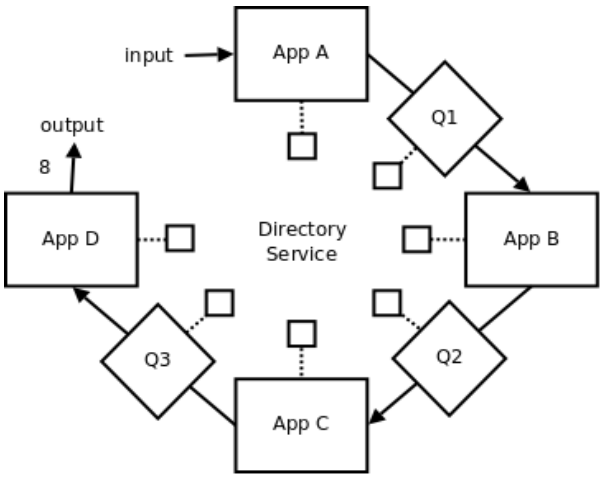
If you are interested in details, ØMQ *chat* example (see tutorial) is an implementation of "distributed broker" architecture.

Distributed directory service

In some deployments it is imperative to avoid single point of failure. In other words, if one subsystem is destroyed or fails, other subsystems should continue working. While previous model is completely distributed message-wise, its configuration is still centralised in the directory service. If directory service fails or when it is unaccessible, system fails.

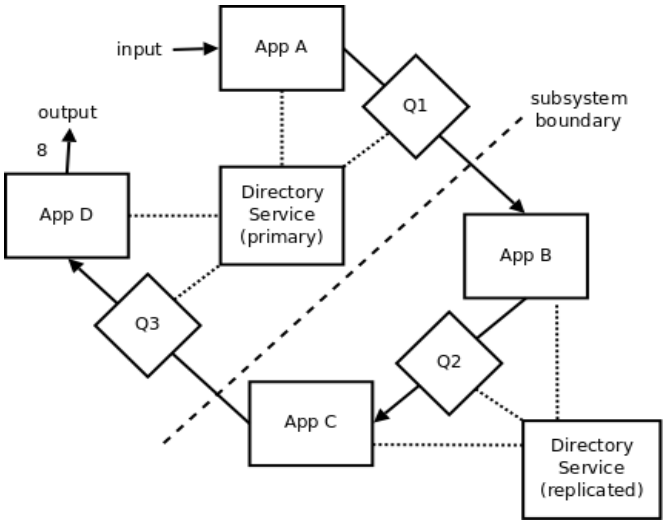
To solve this problem we need a distributed directory service. Simplest example is a production line. While it is developed developers can rely on centralised directory service. Single point of failure is not an issue during the development. Once the production line is deployed, we can copy the configuration to all the nodes of the network. The idea is that once deployed the network topology of production line will be completely stable and thus the issue of having to modify configuration on all the nodes is irrelevant.

Following picture shows this kind of architecture (small empty squares represent the copies of configuration):



Still, many environments require both no single point of failure and dynamically configurable network topology. Think of a large bank. Failure of a single node (centralised directory service) shouldn't stop all the processing in the bank. Even if a branch office goes completely offline, processing in the office shouldn't stop. On the other hand, the networking topology in the bank is constantly evolving. New computers are bought, old ones are dumped. New software services are deployed, old ones are discarded. New network links are being established etc.

In this case there's a need for real distributed directory service. An example may be LDAP service, presumably already installed in the bank. Such a service supports replication - meaning that the configuration is available in the branch office even if it goes offline as there is replicated LDAP server at the place. Following picture shows the architecture described:



Conclusion

While traditional messaging systems tend to use one of the models described above ("broker" model in most cases) ØMQ is more of a framework that allows you to use any of the models or even combine different models to get the optimal performance/ functionality/ reliability ratio.