220

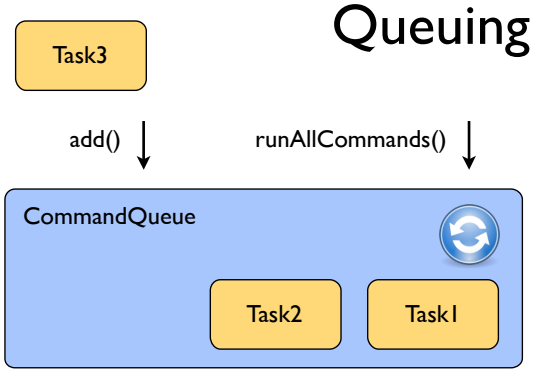
Publish/Subscribe

Dr Robert Chatley - rbc@imperial.ac.uk

@rchatley **#doc220**

In this lecture we look at an architectural style known as publish-subscribe (or pub-sub). We examine it different uses and configurations, and compare it to some of the structures and patterns that we have already seen in the course.

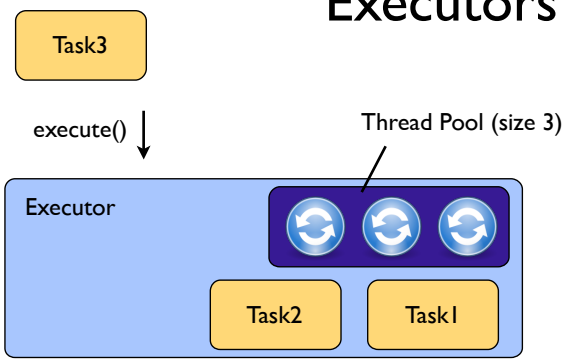
Queuing



#doc220

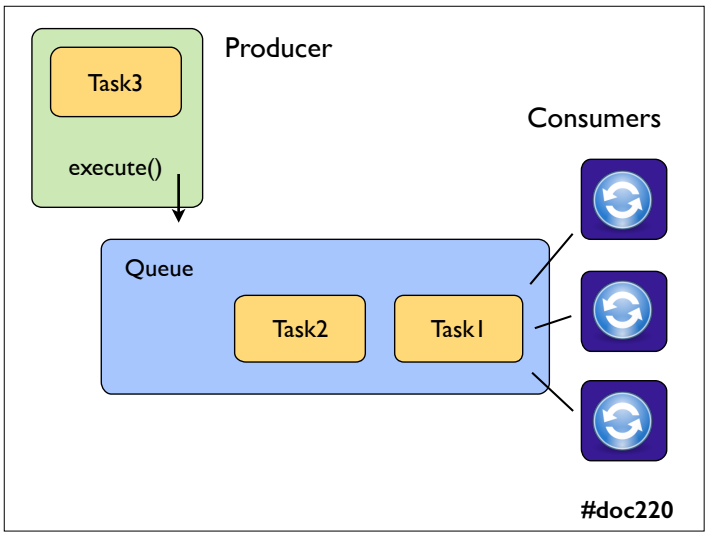
In the previous lecture we looked at ways of queuing up objects that represented units of work, and having them execute later. All in the context of a single thread, we add tasks to the command queue, and then later tell the CommandQueue to execute all the commands that it is currently holding, in order, one at a time.

Executors

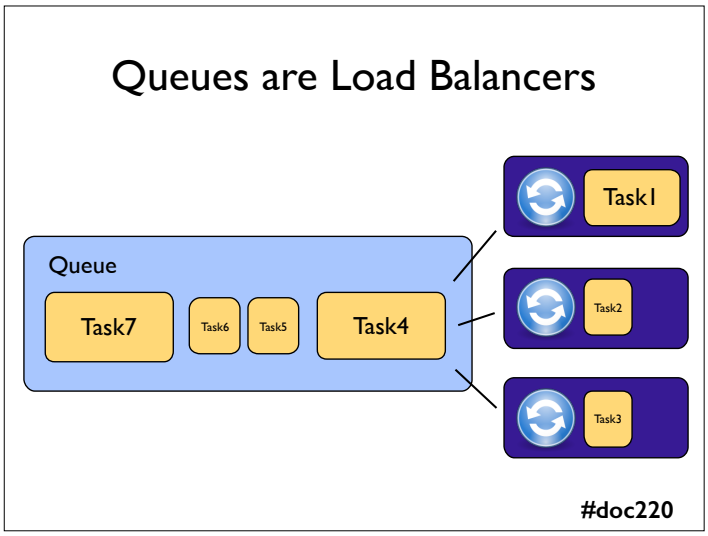


#doc220

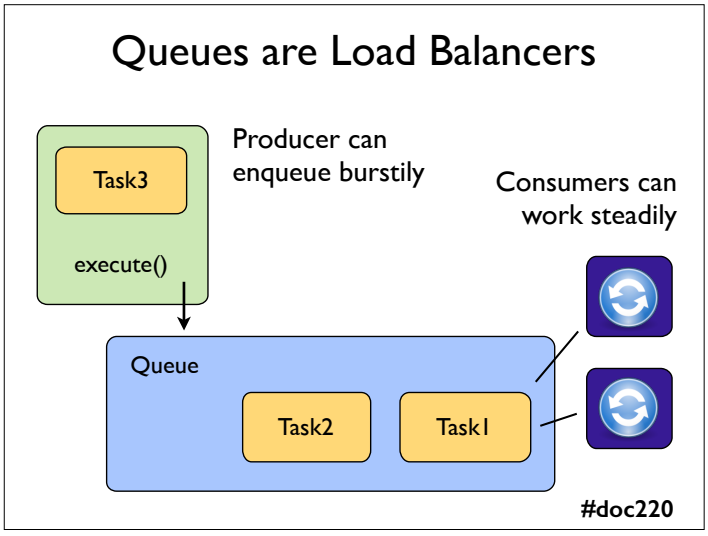
We also looked at how we could exploit the possibility of parallelism by using more threads to execute the queued tasks. If instead of our CommandQueue, we use an Executor, the Executor can manage a pool of threads, and use these to execute our tasks concurrently. When one of the threads becomes free, it can take a new task from the queue. We will see how this generalises to a producer-consumer model, and publish-subscribe.



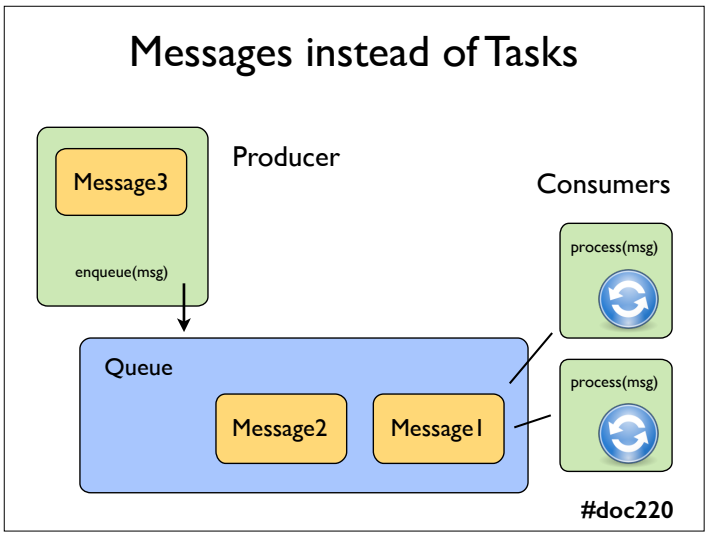
Here we see the basic concept of producers and consumers. Producers create work items, tasks, and put them onto the queue for someone else to process. The consumers monitor the queue and if there is a task there they will pick it up. The notion of a queue is one of the key concepts in publish-subscribe architectures, together with producers and consumers. In this diagram we see multiple consumers consuming from the same queue.



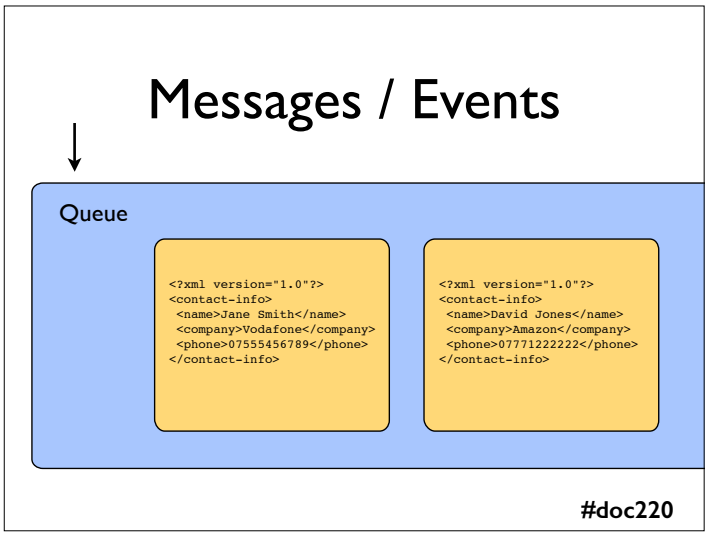
When multiple consumers consume from the same queue, each work item is delivered to exactly one consumer. In this way the queue acts as a load balancer. We can enqueue tasks of different sizes, and the next consumer to become free will pick up the next task. If a short task is finished quickly, that consumer will be ready to take a new task sooner, while maybe another consumer is busy processing a long-running task. The work is balanced between the consumers.



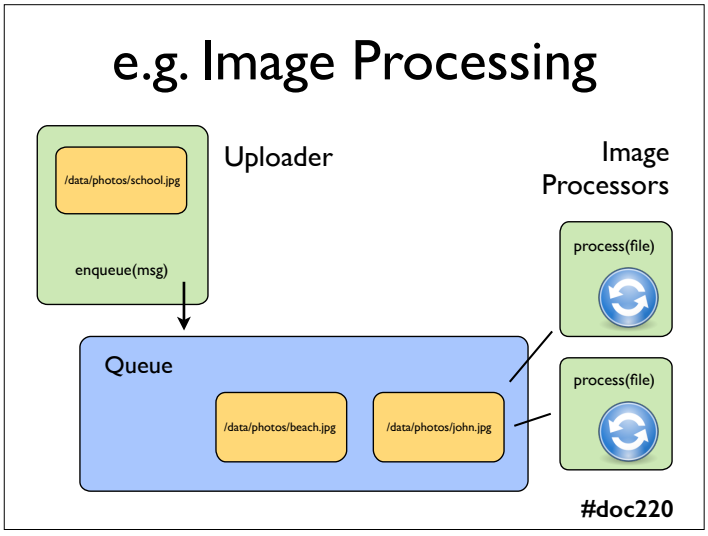
The queue can also act as a regulator or buffer for incoming work. Say a load of users suddenly put jobs into our system, we may not be able to handle them in real time, but we can queue them up and do them as fast as we can, recovering later when the arrival rate of work is lower. The queue helps to decouple the arrival rate of work from the rate at which it can be completed.



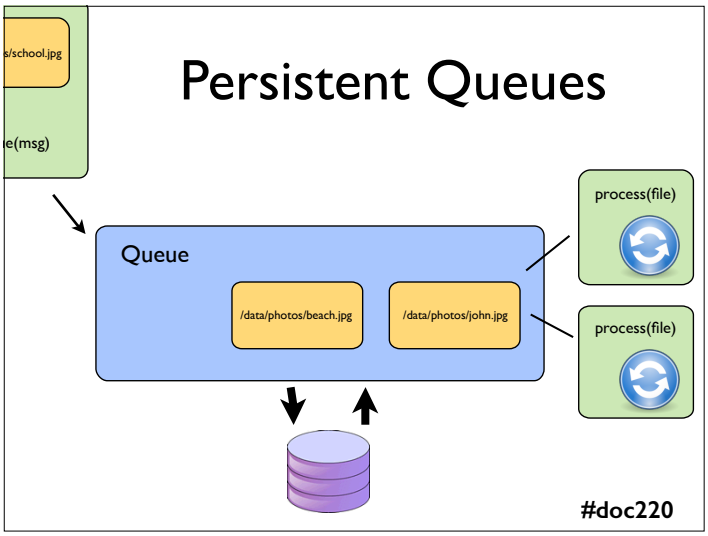
So far we have looked at queues that hold executable tasks - objects - and in the case of an Executor either a Runnable or a Callable. It is quite common for us to put data, rather than executable objects, onto queues. These are generally known as *messages*. Then we can implement any type of consumer - it doesn't have to be a thread, and we can write it separately from the producer. The implementations don't have to be coupled, as long as they understand the same data format.



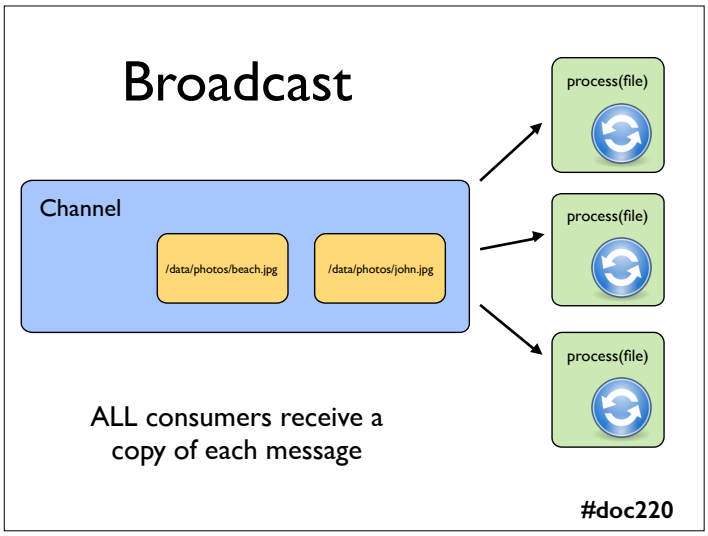
Here is an example where we have put some pieces of data into a queue to represent contact details for two people. Maybe we're going to sync these to an address book, or maybe our application needs to send them a text message. We've represented the data in a simple form (XML) that is both human and machine readable, and can be processed as a string.



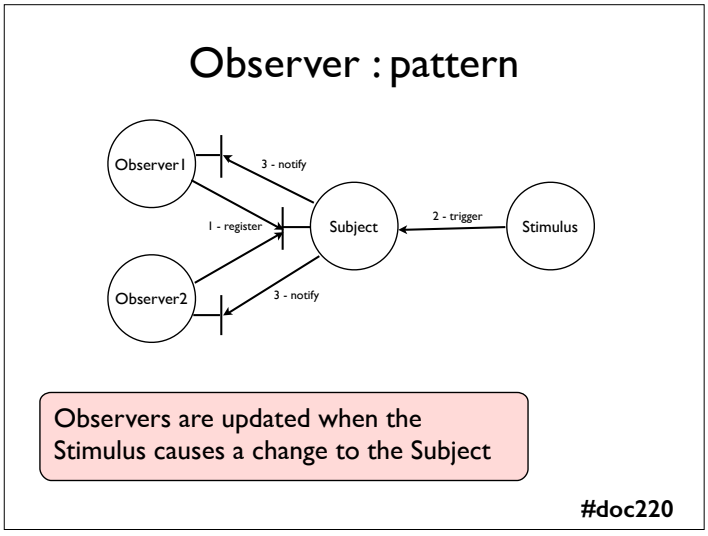
Here is another example where we deal with processing images. Say we are writing a photo album website and people can upload photos. When people upload photos we want to resize them to thumbnails, and maybe do some other processing on them. We put messages on the queue with the file paths of the new photos. Then the consumers can pick these up one at a time and do the image processing. In the mean time we can get back to the user and say "photo processing in progress".



Another feature of some queue implementations is that we can make them persistent. Each message that is put on to the queue is stored on disk or in a database until it is consumed. This is useful in situations where it is very important not to lose messages, and the component implementing the queue might be restarted at various times - or the machine it is running on could be rebooted or fail.



There is another pattern associated with publish-subscribe, where instead of making sure that we deliver each message to exactly one consumer, we broadcast the message to everyone who is listening. Going back to our image processing example, we might have three different jobs that need to be done for each photo that is uploaded, maybe creating a thumbnail, creating a black-and-white version, and checking for friends' faces. We can have three separate consumers all receive the same message and process it in different ways.



This interaction is characterised by the Observer pattern. We will talk more about this pattern in future in different contexts. Many consumers (observers) can subscribe to messages from the producer. The producer doesn't know how many consumers there are; it sends its messages to anyone who wants to listen. The broadcast channel just decouples the producer even more from the consumer. Also many producers may publish to the same channel.

Observer

Your object needs to know when another object's state changes.

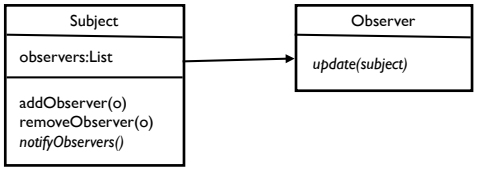
Don't keep asking "Are we nearly there yet?", just ask to be told when you arrive.

This is also known as "register a callback", or "publish-subscribe"

#doc220

This model, where we do something in response to an event or a notification, is known as the Observer pattern, which is described in the Gang of Four book. We invert the control of the program, so that instead of calling the message channel to ask whether it has a new message, we register an object to be notified when something of interest happens. We can register a number of observers, all looking at the same subject, and each gets informed of changes.

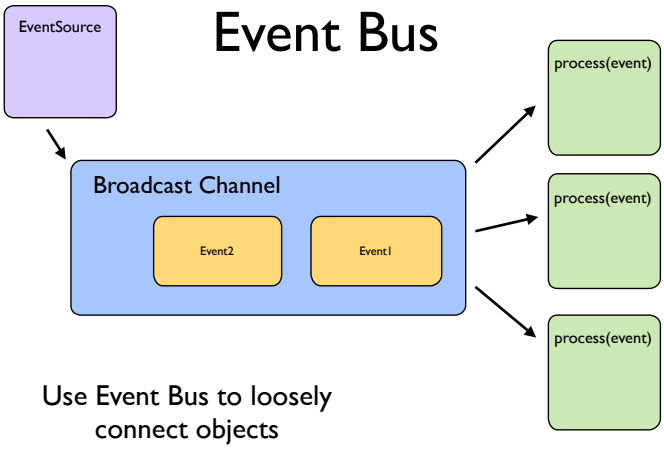
Observer : Example



#doc220

The Observer has two participants: the subject, where we can register observers, and the observer, that will be notified when there is a change of state that is of interest. The subject loops through all of its registered observers when there is a change of state, and sends an event to each observer.

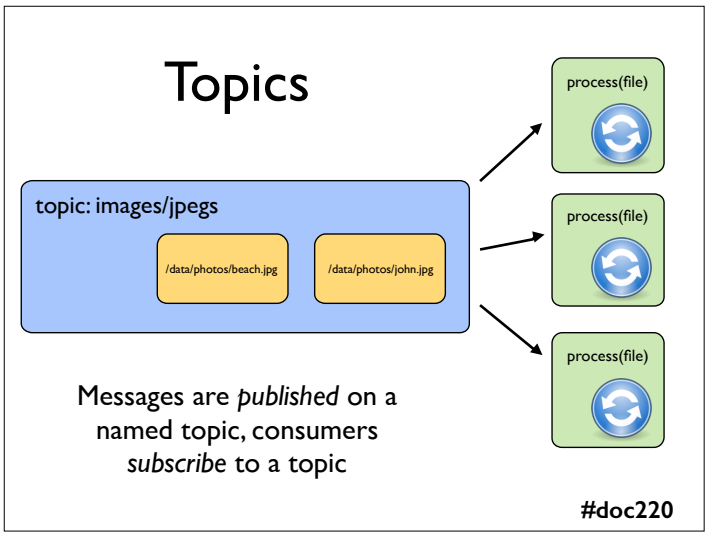
Event Bus



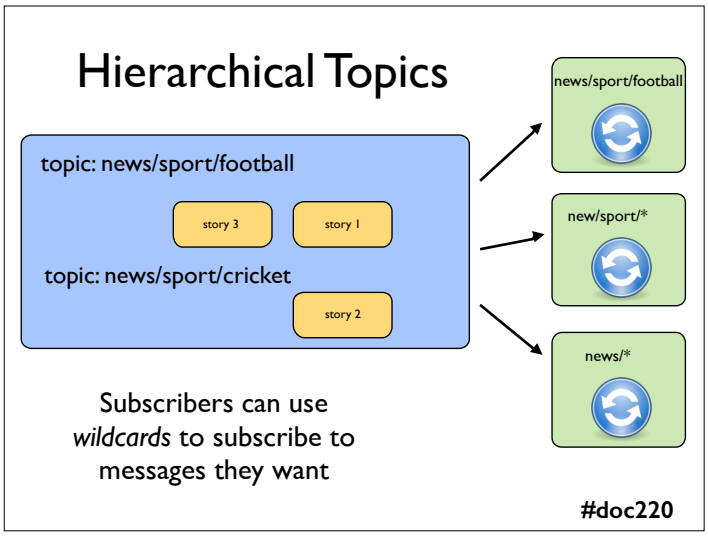
Use Event Bus to loosely connect objects

#doc220

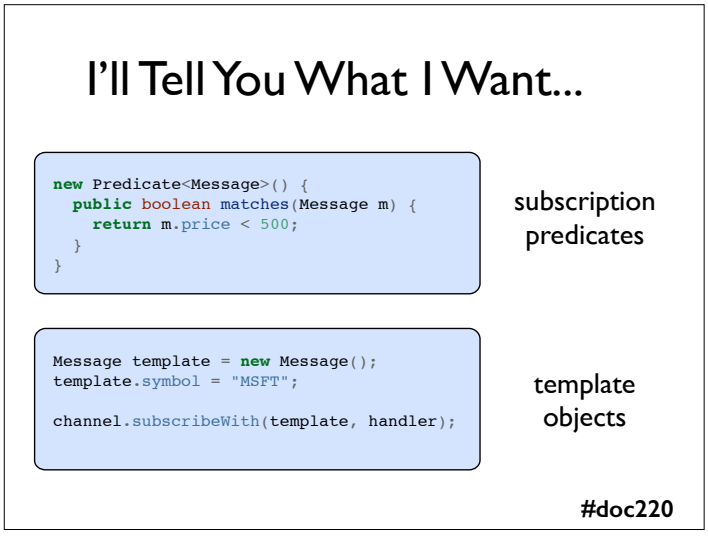
We can use this model to provide a communication mechanism between objects in our application without all of them being tightly coupled together. Instead of calling methods directly on other objects, we can publish events to a broadcast channel known as an Event Bus. Then any other objects that are interested can subscribe to the Event Bus, receive the messages and act accordingly. We can add a new subscriber without having to modify the publisher.



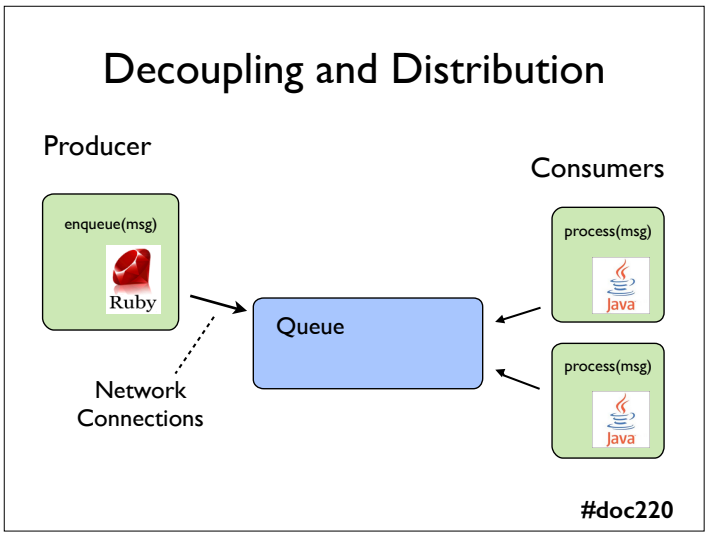
Sometimes a subscriber is not interested in all of the messages, but only a subset. We can aid this situation by publishing messages on a named *topic* inside the broadcast channel. For example here we publish these image file paths to the topic images/jpegs. If a subscriber is only interested in RAW files, it could listen to a different topic.



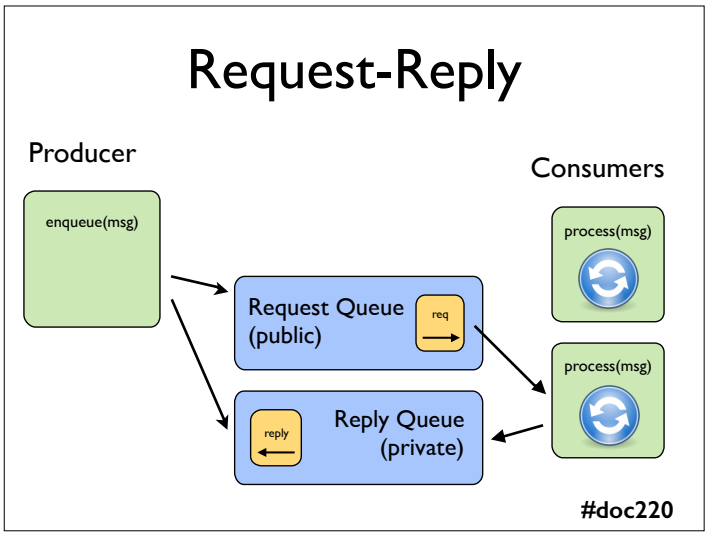
We could also create hierarchical topics, to allow subscribers to detail how much of the message stream they are interested, perhaps using a wildcard. Here we are publishing news articles, and one consumer subscribes only to articles about football, one to all sport articles, and one to all news articles (including sport, which includes football).



There are other ways of filtering the messages that a consumer subscribes to. Here we show a couple of alternatives. One uses a Predicate object defining a filter to match messages. We could build a system where the subscriber registers their subscription with the channel passing this predicate, and the channel would use this to decide which messages to deliver. Alternatively we could provide a template message - an object which is partially populated - to match against.



Sometimes we send a message on a queue and it may be picked up by another process, possibly written in a different language, maybe running on another machine. If we only send data in our messages, then queues and topics become a useful way of distributing work and communicating between machines. We send our messages over the network rather than passing them between local objects.



So far we have sent messages and assumed that they will be processed however is appropriate - but sometimes we want a reply (especially if we are using queues and topics to communicate with another system running on another machine). We can do this using two channels. We publish our request on a public channel, for any consumer to pick up. Then, when that consumer has processed it, it sends a message back on the private channel to which only the original sender subscribes.

When we start to using messaging to communicate between processes and machines, it is useful to have an external, purpose built broker to handle the routing of messages. There are many different products available that do this with different levels of performance - and cost.