Systems Integration

Often a complete solution for a media production and delivery workflow relies on more than one vendor's product. Whatever capability each of these products have, to achieve an effective solution, these systems will undoubtedly need to share data as well as media. Although simple 'file drops' of media and data files (typically xml) works in many cases, a more elegant solution is often required using the various systems APIs (Application Programming Interfaces).

Many APIs work by web services or similar, where one system makes a query to another system holding data, and that system returns the data based on that query criteria. This approach works well so long as the requesting system knows what data record it is looking for, but in many cases, the requesting system needs to know about new records as they are created. One approach to this is to constantly poll the system holding data, making a query such as 'Tell me about all new data since a certain time". There are several drawbacks to this approach: Firstly, the requesting system will need to make a request regularly, and even if there are no new records to report, both the requesting system and the system holding data will have a process overhead by these requests – a waste of processing and network bandwidth. Secondly, there may be many new records to report, with both systems potentially being overloaded by the shear amount of data to process. Finally, if the poll time and the query since time are not carefully set, duplicate records will be returned, or worse, records could be missed.

A better method is for the system holding the data that the requesting system needs to know about to notify the requesting system of changes or new records. For example, with a media asset management system (MAM), for each new media asset that is created, the MAM notifies subscribers of changed or new asset's unique ID. If the subscribing system has a requirement for such data (extra data fields such as media type or category could also be sent, in order for the external system to know whether it is interested or not), the subscribing system can then make a full request through the MAM system's web services API to obtain a complete record. Such a notification method is described in this document.

It should be noted that the terminology used in this paper is that of a specific message broker: Apache ActiveMQ. Other message brokers offer similar, and in many cases, enhanced capabilities, and often use different terminology to describe similar features and functionality.

Message Broker

It is highly likely that different systems will be built using different software platforms and will use different methods for communication, especially around 'near real-time' notification services. It is also quite likely that requesting systems will have many other tasks to perform, and so will not be able to immediately react to a notification message. Due to both of these requirements, a 'middleware' service called a message broker is often used.

Message brokers provide several services: First, they hold a '**message queue**' where a '**provider**' system places notification messages, from which a '**consumer**' takes the next message from the queue when it is ready to do so. Because the message broker is a separate service application, the provider and consumer

can both work at their own pace. If the consumer is for some reason slower at taking messages from the queue than the provider is at placing them on the queue, the queue will grow until the consumer catches up. Typically, when the consumer requests the next message, it will be sent the oldest to be placed there (first in – first out). Queues can grow to any length until the message broker service runs out of memory. For this reason, it is important that the queue is checked (especially during system development) to make sure it is not getting too long, and that messages are kept as short as functionally. Some message broker systems have special system queues that can notify consumers that would then report to users that there is a queue building.

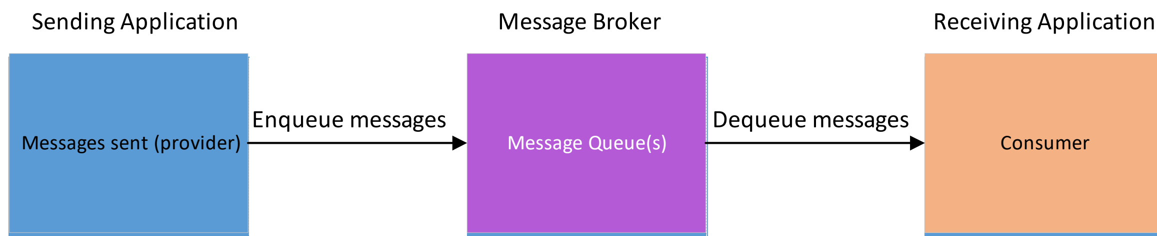

Fig.1, Basic Message Broker system: providers enqueue messages, consumers dequeue messages

The diagram above shows a basic message broker system. The 'provider' adds a new message to the queue every time it needs to communicate something to the 'consumer'. The message broker will hold on to the message (and any subsequent messages) until the consumer is ready to 'de-queue' the message. The consumer can only de-queue the message (remove it from the queue). It cannot 'peek' at the message to see its content and leave it there.

It is possible (but not typical) to have more than one provider (if there are multiple message sources needed, it is easier to manage multiple queues, one for each purpose).

It is possible (and typical) to have more than one consumer. However, since the first consumer to de-queue the message will remove it from the queue, only that consumer will receive that message. Other subscribed consumers will not see that message. This may seem like an odd behaviour (we'll see how 'topics' can help us with multiple consumers later), but for a consuming system that needs resilience this can be extremely useful. If the consuming application is resilient, it can have multiple consumer services all pointing at the same queue. The first operable consumer service will de-queue (and subsequently process that message), with others not processing the same message. This avoids duplication of processing and avoid duplicate records in the consuming system. If a consumer service fails, then the next available service will be able to pick up the next message.

Message data can be any data, but typically text based xml is used, and messages are kept as short as possible in order to maximise efficiency. The message will often carry additional data such as provider information, a unique ID and other data such as type of message. None of this additional data is essential for a basic messaging system to function.

Queues vs. Topics

As we saw above, message broker queues typically have a one to one relationship: that is, a provider places a message on a queue and one (and only one) consumer de-queues the message.

Often it is desirable for multiple consumers from different systems to be notified with the same message. The provider could send to multiple queues (which has the advantage of making debugging easier as it is easy to spot which consumer has stopped consuming - the queue for that consumer will grow), however this can become cumbersome to manage if there are many consuming systems.

A 'Topic' is a type of queue specifically designed to have multiple consumers. Providers send messages to Topics in the same way as they send messages to regular queues. Consumers 'subscribe' to a Topic, and the message broker keeps a record of subscribing consumers. When a new message is added to the queue, the message broker will keep a copy of that message until all subscribed consumers have read the message. Only when all consumers have read the message will the message be deleted from the topic. Consumers must remain subscribed to a topic in order to receive messages. If they drop the connection (or unsubscribe), the message broker will no longer hold messages from them.

Topics can be a useful way of sanding the same message to multiple subscribers, there is however a drawback that if any subscribed consumer stops processing messages (but is still subscribed), the topic queue will grow.

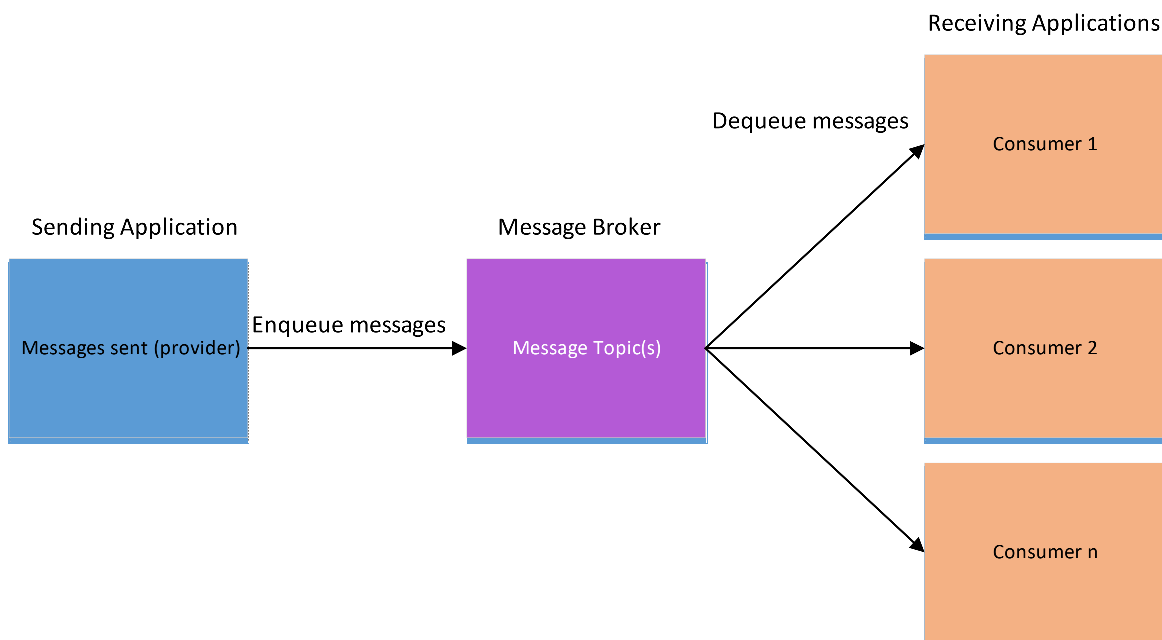

Fig.2, Message Topic: providers enqueue messages, consumers subscribe to topics and all must dequeue

System Integration

Message brokers are used between systems where one system notifies other systems of changes to data records. A common example of this being used in the media and broadcast industries, is when a media asset management system (MAM) needs to update other systems of asset changes (e.g. informing a planning and scheduling system of changes to assets). Most MAM systems provide a service to allow external systems to obtain data on a specific asset. However, most of these work by the external system requesting the data for a specific asset, or make a search request to return a number of assets meeting a criteria. A more efficient way of handling this interaction is by use of a message broker.

The MAM system (the provider in this case) enqueues a message to the message broker. The message may contain only an ID of the asset that has changed (a very short message). The scheduling system (the consumer in this case) dequeues the message. If the message contains an ID that is of interest, then a request is made through whatever API has been made available by the MAM (typically web services or SOAP) to obtain the full details of that asset (could be significant amounts of data). The diagram below shows the sequence of transactions (time flows from top to bottom)

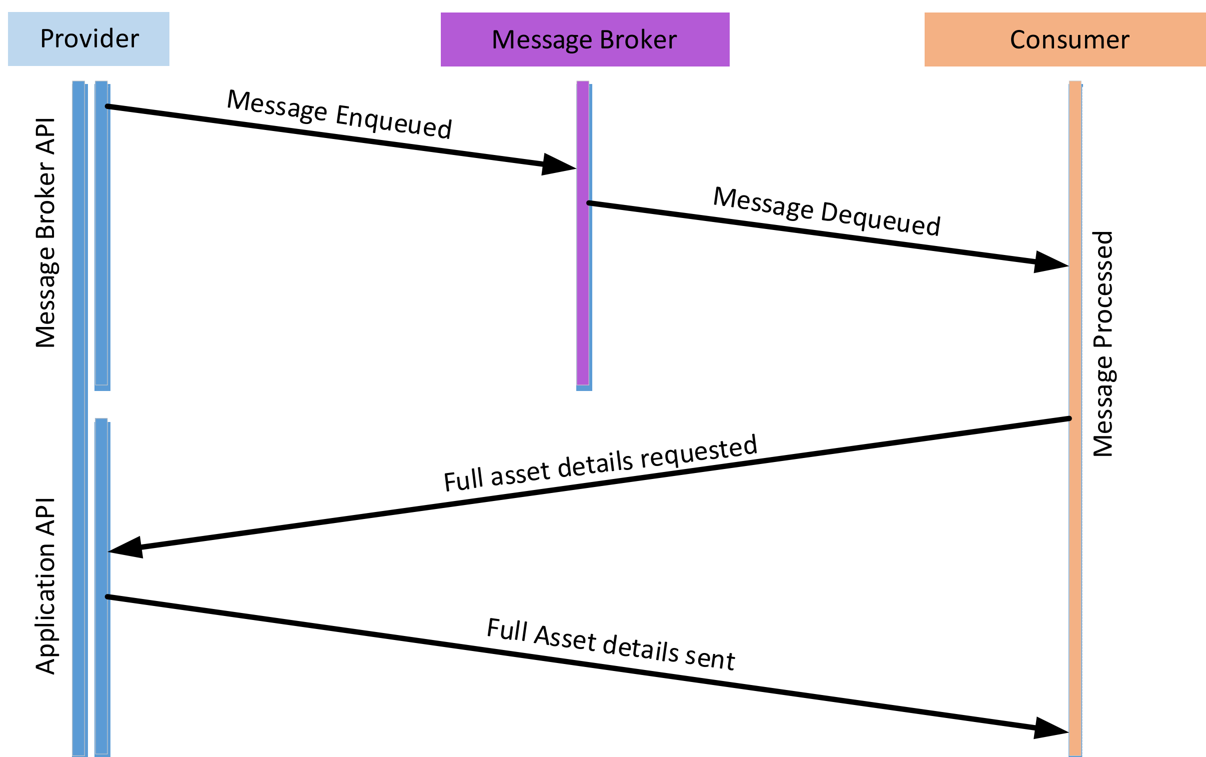

Fig.3, Message queue used for notifications to consuming application, which then requests full asset details