Status Update Design

Status Update

Following on from the high level design, this page will provide a more detailed design approach to implement status update logging in the Java broker.

The logging hierarchy identified is not suitable to be directly used by Log4j as there is multiple routes and loops in the graph.

Abstracting the logging is recommended as this will allow us to simply provide Qpid specific optimisations such as providing the log prefix.

This design will cover the following areas:
Contents

Additional Documentation

- Logging Format Design
- Functional Specification
- Test Plan
- Test Specification
- Technical Specification

Logging Configuration

At this stage configuration will be limited to the addition to the main config.xml file of the following option:

```
<broker>
 ...
  <status-updates>ON</status-updates>
 ...
</broker>
```

This *status-update* on setting will be the default value if the section is not included in the configuration file. The setting will be global for all virtualhosts and will be exposed via the management console as logger 'qpid.status' to allow dynamic setting.

The ability to configure more fine grained logging will be investigated here, but will not be implemented in the initial phase.

Status Updates

In the first phase updates only status updates will be provided. Status updates should take the form of general operational logging level, no logging on message delivery path way and No performance impact. The recommendation will be to have these enabled for production use.
e.g. Creation/Destruction events

The status updates can also be used in a second phase to provide additional logging to assist development.
The additional logging can be performed on the message delivery path way. This may have performance impact and so would not be recommended for long term production use.
e.g. Message Enqueue/Dequeue

Logging Abstraction

The abstraction layer allows us to fully decouple the logging mechanism from any operational logging that the broker may wish to perform. The following code highlights show how we would abstract the logging operations.

The approach to logging is that a *LogActor* will be recoreded as a *ThreadLocal* and will be used to perform logging the log messages as highlighted on Logging Format Design. The *LogActor* will take two parameters, the *LogSubject* and the *LogMessage*. When a Status event occurs that should be logged the *LogActor* can be retrieved from the thread thus avoiding passing the *LogActor* through as a parameter to all locations were it must be logged. The initial *LogActors* will be **AMQPActor** and **ManagementActor**. Later phases would introduce **HouseKeepingActor**. These *LogActors* are responsible for checking that the logging should be performed for both themselves and the *LogSubject*. The *LogActor* then provids their log formatted name as per the format design along with the message to the *RootMessageLogger*. Initially the configuration will be a simple on/off, however, in a future phase the details can be used to identify if logging should proceed for that *LogActor* and *LogSubject* combination. At this stage selective configurations is not part of this design.

The use of the *LogActor* allows for situations such as *Binding* to have a *Connection* associated with the *Binding*. This will allow a *Binding* create event to be logged like this:

```
2009-06-29 13:35:10,1234 +0100 MESSAGE [con:1(guest@127.0.0.1/)/ch:2] [ex(amq.direct)/qu(testQueue)/bd
(routingKey)] BND-1001 : Binding Created
```

rather having no details about how the creation occurred:

```
2009-06-29 13:35:10,1234 +0100 MESSAGE [ vh(/)/ex(amq.direct)/qu(testQueue)/bd(routingKey) ] BDN-1001 : Create
```

Interfaces

LogActor

```
/**
 * LogActor the entity that is stored as in a ThreadLocal and used to perform logging.
 *
 * The actor is responsible for formatting its display name for the log entry.
 *
 * The actor performs the requested logging.
 */
public interface LogActor
{
    /**
     * Logs the specified LogMessage about the LogSubject
     *
     * Currently logging has a global setting however this will later be revised and
     * as such the LogActor will need to take into consideration any new configuration
     * as a means of enabling the logging of LogActors and LogSubjects.
     *
     * @param actor   The actor that is requesting the logging
     * @param message The message to log
     */
    public void message(LogSubject subject, LogMessage message);
}
```

LogSubject

```
/**
 * Each LogSubject that wishes to be logged will implement this to provide their
 * own display representation.
 *
 * The display representation is retrieved through the toString() method.
 */
public interface LogSubject
{
    /**
     * Logs the message as provided by String.valueOf(message).
     *
     * @returns String the display representation of this LogSubject
     */
    public String toString();
}
```

RootMessageLogger

```
/**
 * The RootMessageLogger is used by the LogActors to query if
 * logging is enabled for the requested message and to provide the actual
 * message that should be logged.
 */
public interface RootMessageLogger
{
    /**
     * Determine if the LogSubject and the LogActor should be
     * generating log messages.
     *
     * @param logSubject The subject of this log request
     * @param logActor   The actor requesting the logging
     * @return boolean true if the message should be logged.
     */
    boolean isMessageEnabled(LogActor actor, LogSubject subject);

    /**
     * Log the raw message to the configured logger.
     *
     * @param message   The message to log
     * @param throwable Optional Throwable that should provide stact trace
     */
    void rawMessage(String message, Throwable throwable);
}
```

RawMessageLogger

```
/**
 * A RawMessage Logger takes the given String and any Throwable and writes the
 * data to its resource.
 */
public interface RawMessageLogger
{
    /**
     * Log the message and formatted stack trace for any Throwable.
     *
     * @param message   String to log.
     * @param throwable Throwable for which to provide stack trace.
     */
    public void rawMessage(String message, Throwable throwable);
}
```

Logging Usage

Logging of a Channel Creation

```
pubic class Connection
...
   LogActor amqpActor = // retrieved from ThreadLocal.
   //_channelSubject is an instance LogSubject that knows how to represent this Connection
   amqpActor.logMessage(_connectionSubject, LogMessages.CHANNEL_CREATE(this));

...
```

Would result in the following based on the Logging Format Design.

```
2009-06-29 13:35:10,1234 +0100 MESSAGE [con:1(guest@127.0.0.1/)] [ch:2] ChM-1001 : Channel Created
```

Logging of a new consumer creation

```
...
        amqpActor.logMessage(_subsriptionSubject, LogMessages.SUBSCRIPTION_CREATE(this));
...
```

Would result in the following:

```
2009-06-29 13:35:10,1234 +0100 MESSAGE [con:1(guest@127.0.0.1/)/ch:2] [sub:1:qu(myqueue)] Sub-1001 :
Subscription Created
```

Initial Status Messages

Broker

Startup
Configuration details
Ready
Shutdown

ManagementConsole

Startup
Configuration details
Ready
Close

VirtualHost

Create
Configuration details
Close

MessageStore

Startup
Recover Status
Start
Progress
End
Close

Connection

Open
Close

Channel

Create
Flow Status
Destroy

Queue

Create
Destroy

Exchange

Create
Destroy

Binding

Create
Destroy

Subscription

Create
Destroy