MQTT

MQTT is a machine-to-machine (M2M)/"Internet of Things" IOT connectivity protocol.

WHAT? WHERE? WHY? POWER? TRAFFIC? RESPONSE?

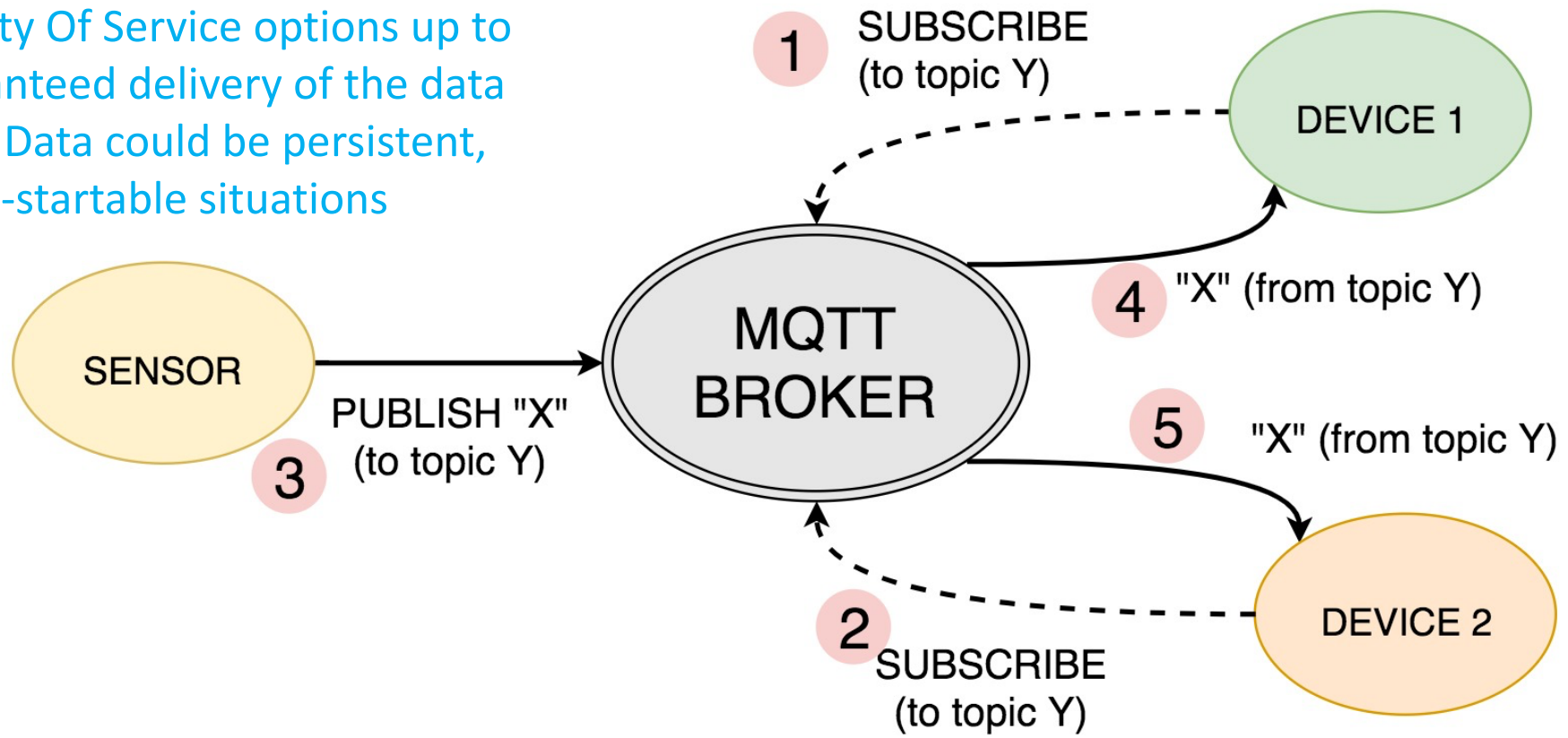
@ LabWare ESUG Aug 2019

By John M McIntosh

Blob of Data & a topic string

MQTT – Simplistic (Subscribe & Publish: 1 to many)

Quality Of Service options up to guaranteed delivery of the data blob. Data could be persistent, for re-startable situations



https://www.norwegiancreations.com/2017/07/mqtt-what-is-it-and-how-can-you-use-it/

MQTT – Power/Traffic & FaceBook?

Dr Google:

- Up to 22% more energy efficient & 15% faster than HTTP.
  Does not depend on if the connection type is 3G or WiFi.

It is the transport layer for FaceBook Apps & a serious platform for IBM

IOS libraries are mature, *Android not so much*.

MQTT is now at Version 5 (our winter 2020 work)

Implemented Version 3.11, we even found a bug in the spec!

*Specs are 50ish pages, other protocols > 100 pages.

MQTT – Various Smalltalk Projects about

Took Tim Rowledge's MQTT **client** for Scratch on the PI.

http://www.squeaksource.com/MQTTClient.html

Various fixes fed back to Tim

Need Data Broker in Smalltalk? Why?

A commercial broker would invoke customer verification & validation procedures.

But we can add a MQTT data broker to a VSE image as an SLL! {Software upgrade}

Set out to refactor a client & create a data broker solution in pure Smalltalk.

Extra make a platform MQTT client in Scarlet SmallTalk/swift/java

Pharo version at https://github.com/LabWare/MQTT-broker MIT License

MQTT- Simple? Oh sure a month or four…

Yes, the standard is simple? Just grind thru the specs…

#	MQTT Broker/Client Standard and our solution & work notes
[MQTT-1.5.3-1]	The character data in a UTF-8 encoded string MUST be well-formed UTF-8 as defined by the Unicode specification [Unicode] and restated in RFC 3629 [RFC3629]. In particular this data MUST NOT include encodings of code points between U+D800 and U+DFFF. If a Server or Client receives a Control Packet containing ill-formed UTF-8 it MUST close the Network Connection.
[MQTT-1.5.3-2]	check for subscribe, unsubscribe topic A UTF-8 encoded string MUST NOT include an encoding of the null character U+0000. If a receiver (Server or Client) receives a Control Packet containing U+0000 it MUST close the Network Connection.
[MQTT-1.5.3-3]	A UTF-8 encoded sequence 0xEF 0xBB 0xBF is always to be interpreted to mean U+FEFF ("ZERO WIDTH NO-BREAK SPACE") wherever it appears in a string and MUST NOT be skipped over or stripped off by a packet receiver.
[MQTT-2.2.2-1]	Where a flag bit is marked as “Reserved” in Table 2.2 - Flag Bits, it is reserved for future use and MUST be set to the value listed in that table.
[MQTT-2.2.2-2]	If invalid flags are received, the receiver MUST close the Network Connection.
[MQTT-2.3.1-1]	**MQTTTransportLayer>>newPackID** SUBSCRIBE, UNSUBSCRIBE, and PUBLISH (in cases where QoS > 0) Control Packets MUST contain a non-zero 16-bit Packet Identifier.
[MQTT-2.3.1-2]	**MQTTTransportLayer>>badPacketID** & **newPacketID** Each time a Client sends a new packet of one of these types it MUST assign it a currently unused Packet Identifier.
[MQTT-2.3.1-3]	**MQTTPacketPublish>>acknowledgement, MQTTPendingPubCompJob>>resendFor:ifNeededAtTime: MQTTPacketPubRel>>acknowledgement, MQTTTransportlayerServer>>handleSubscribePacket:** If a Client re-sends a particular Control Packet, then it MUST use the same Packet Identifier in subsequent re-sends of that packet. The Packet Identifier becomes available for reuse after the Client has processed the corresponding acknowledgement packet. In the case of a QoS 1 PUBLISH this is the corresponding PUBACK; in the case of QO2 it is PUBCOMP. For SUBSCRIBE or UNSUBSCRIBE it is the corresponding SUBACK or UNSUBACK.
[MQTT-2.3.1-4]	**MQTTTransportLayerServer>>handlePublishResponse:** The same conditions [MQTT-2.3.1-3] apply to a Server when it sends a PUBLISH with QoS >0.

MQTT – Testing

IBM MQTT compliance test scripts, plus tests via other vendors.

Need more test data, sure what if we do **24 GB a day of data**.

https://test.mosquitto.org

Subscribe to wild card topic *, broadcast thru our system, back to a Linux command line client or three… Run for weekend, anything explode?

Oh say that looks that like personal data? Some people have no problem making public test server “their” production server.

Work work - March 2018 –> March 2019

Subscribe and Publish seem simple which makes MQTT a 1 to many abstraction, but we needed a simplified 1 to 1 relationship to hide the complexity from internal engineers as most communications is an entrenched mind set of *client/server*.

Our solution is something we called an

Endpoint.

Endpoint - Hiding complexity

Datum + tag + target. Datum is opaque, tag is meta-data.

Based on this we create a Topic somewhat like:

{Session ID} / {target EP } / {receiver EP} / {Command} / {EP Service} / {Tag} / {promise ID}

- *Private to user 492 {Session ID}*
- *Targeting a particular target Endpoint -> (Mobile) device.*
- *Mobile device looks up Endpoint service executes the Tag and Datum against it.*
- *Response if asked for will be returned to { Receiver Endpoint}*

Endpoint -Send

Three different options:

>>**send:** aByteArray tag: aTagString
“Async send, no response, or receiveError”

>>**sendSync:** aByteArray tag: aTagString
“Sync send, will get response or error data”

>>**sendAsyncResponse:** aByteArray tag: aTagString
“Response via receiveMessage or possible receiveError”

Endpoint target identifier is defaulted as a peer to peer, but can override.

Need to exploit a Promise class to do the right thing for **sendSync:**

Endpoint - Receive

>>receiveMessage: byteArray tag: tag
“incoming message what should the tag do with the *aByteArray* data?
tag == ‘hello’ ifTrue: […].
^ByteArray new “default response”

Note: Any exception thrown is bundled up as a stack trace and returned to the other Endpoint where it would trigger the Error handler logic via the:

>>receiveError: aByteArray tag: aTagString

Promise – Device & JavaScript Hassles

1. Halt the JavaScript thread.
2. Get data on the MQTT network thread.
3. Signal the JavaScript thread?
4. Resume the JavaScript thread.

Sounds Simple?

No: A painful month of self doubt & failures

* *No the JavaScript promise logic did not solve this issue for us.*

Promise – Device & JavaScript Hassles

Solution: **Share the promise logic between JavaScript & platform**.

1. Stop the JavaScript thread via platform semaphore
2. Examine incoming MQTT publish msgs for Promise fulfillment
3. Fulfil the promise on the platform side.
4. Signal the platform semaphore (or timeout)

JavaScript code then does promise value on the JS side, getting the results from the platform side. In this case we either have data or an error {possible walk back stack}

```
Promise>>waitForMs: milliseconds
	"wait for results, timeout or resolve with results"
	hasValue ifTrue:[^value].
	self resolveWith: (self primWaitForTimeOut: milliseconds).
	^value

Promise>> primWaitForTimeOut: timeOut
   <primitive: 'primWaitForTimeOut' module: 'NKPromise'>
   "value or nil(timeout)"
   self primitiveFailed
```

Scarlet Smalltalk code for Promise

It’s just a primitive call. Note the standard failure code

‘self primitiveFailed’

```
NKPromiseModule.primWaitForTimeOut= function(receiver, args) {
    var timeOut = args[0].valueOf();
    var handler = receiver["@@nativeClient"];
    if (typeof handler == "object" && typeof timeOut == "number") {
        var returnValue = handler.waitForTimeOut(timeOut); //return an Array
        if (typeof returnValue === "undefined")
                return this.primFailValue;


        for (var i=0; i<returnValue.length; i++) {
                        if (returnValue[i].constructor == Uint8Array) {
                            returnValue[i] = smalltalk.ByteArray.contents_(returnValue[i]);
                        }
        }
        return returnValue;
    }
    return this.primFailValue;
}
```

Scarlet Smalltalk JavaScript Primitive for NKPromise

@@nativeHandler is a instance var on receiver that points to the Swift Object reference. Swift obj can point to JavaScript object, JavaScript obj can point to Swift object. **See MVVM**

On return check for nil, then check to see if we have to take a JavaScript Uint8Array object and create a Smalltalk ByteArray object.

ByteArray contents: aUint8Array

```
@objc protocol NKPromiseExport: JSExport {
    func waitForTimeOut( _ timeOut: Int) -> [Any]?
}

@objc class NKPromise: NSObject, NKPromiseExport {
   let waitSemaphore = DispatchSemaphore(value: 0)
   var rawDatum: [Any?] = []

   func resolveWith(_ data: [Any?]) { //MQTT logic or other will resolve promise with data
      self.rawDatum = data
      self.signal()
   }

   func waitForTimeOut(_ timeOut: Int) -> [Any]? {
      let valid = self.waitSemaphore.wait(timeout: timeOut)
      if valid == .timedOut {
         return nil
      }
      return returnDatum()  //Could return ByteArray contents: here, but do in JavaScript caller
   }
```

IOS Swift code for NKPromise

Nasty bits: Allocating memory for JavaScript Objects can only be done on JavaScript Thread, not this MQTT network worker thread. Many threads on the go.

Other complication is that: Who does the garbage collection work, Swift or JS? Maybe 10MB here? Try not to copy it… iOS Easy, Android complicated. We forked Apple's JavaScript Core engine for Android to better address the issue.

```
Endpoint>>sendSync: aByteArray  from: anEndpoint tag: aTagString
receivingManager: receivingManager onError: exceptionBlock  timeOut: aTimeOutInSeconds

    self rememberPromise: (Promise withIdentifier: self nextTransactionNumber).
    estimatedTimeOut := … "Estimate time need to transmit data based on 3G average cost plus 20%"

    [| topic |
       topic := MQTTTopic targetManager: self remoteIdentifier ….
       self publish: aByteArray onTopic: topic] on: Error do: [exceptionBlock].

       “Promise waits only N ms, then raises an error”
    datum := ourPromise waitForMs: estimatedTimeOut onTimeOut: [….]
    metaData := MQTTTopic decodeTopic: datum.
    metaData responseType == #error
        ifTrue:[^EndpointError signal: metaData returnedBytes asString].
    ^ metaData returnedBytes
```

```
Endpoint>>executeBlock
   ^[:t :m :q |
      metaData := MQTTTopic fromString: t.
      receivingManager := metaData receivingManager.
      hasError := false.
      [myEP := self endpointAtRole: myRole partner: myPartner.
       results := myEP perform: metaData selector  with: m with: tag]] on: Error
                  do: [:ex | “Error, return error stack”
                        publishTopic := MQTTTopic targetManager: …
                        self publish: ex description onTopic: publishTopic.
                        hasError := true.
                        ex return: nil].   //Scarlet SmallTalk issue returning an exception value does not work
      (hasError not and: [transportType ~= #async] and: [messageType == #execute])
         ifTrue:[| publishTopic|.
            publishTopic := MQTTTopic targetManager: …  “valid async return value needed”
            self publish: results onTopic: publishTopic].
      transportType == #sync
         ifTrue:[[self executeSyncBlock value: tSession value: m value: q] on:Error […] ]
```

Platform code to JavaScript to find this block to execute.
Earlier did subscribe: topic do: [Block]

Endpoint – Non Obvious things

- *LIMS SaaS user can print to Bluetooth or TCP/IP printer that is visible to device, but not to the SaaS server. {a channel thru customer firewall}*
- *LIMS Server can ask a dedicated device (LIMS Appliance) to poll for file system (NFS) data, then supply the data to the LIMS server. {a channel thru customer firewall}*
- *Device can talk to another device directly via the broker.*
- *MQTT Data Broker ping/pong helps take down zombie sessions.*
- *MQTT reconnect helps with iffy cellular data connectivity.*
- *MQTT guaranteed message delivery can solve 2 phase commit issues.* ***Hard Concept***

Endpoint – 1 to 1 communications