MQ Telemetry Transport

Overview

IBM MQ Telemetry Transport (MQTT) is a lightweight messaging protocol that is designed for Internet of Things (IoT) and mobile connectivity.

Jump to:

Getting started with MQ Telemetry Transport

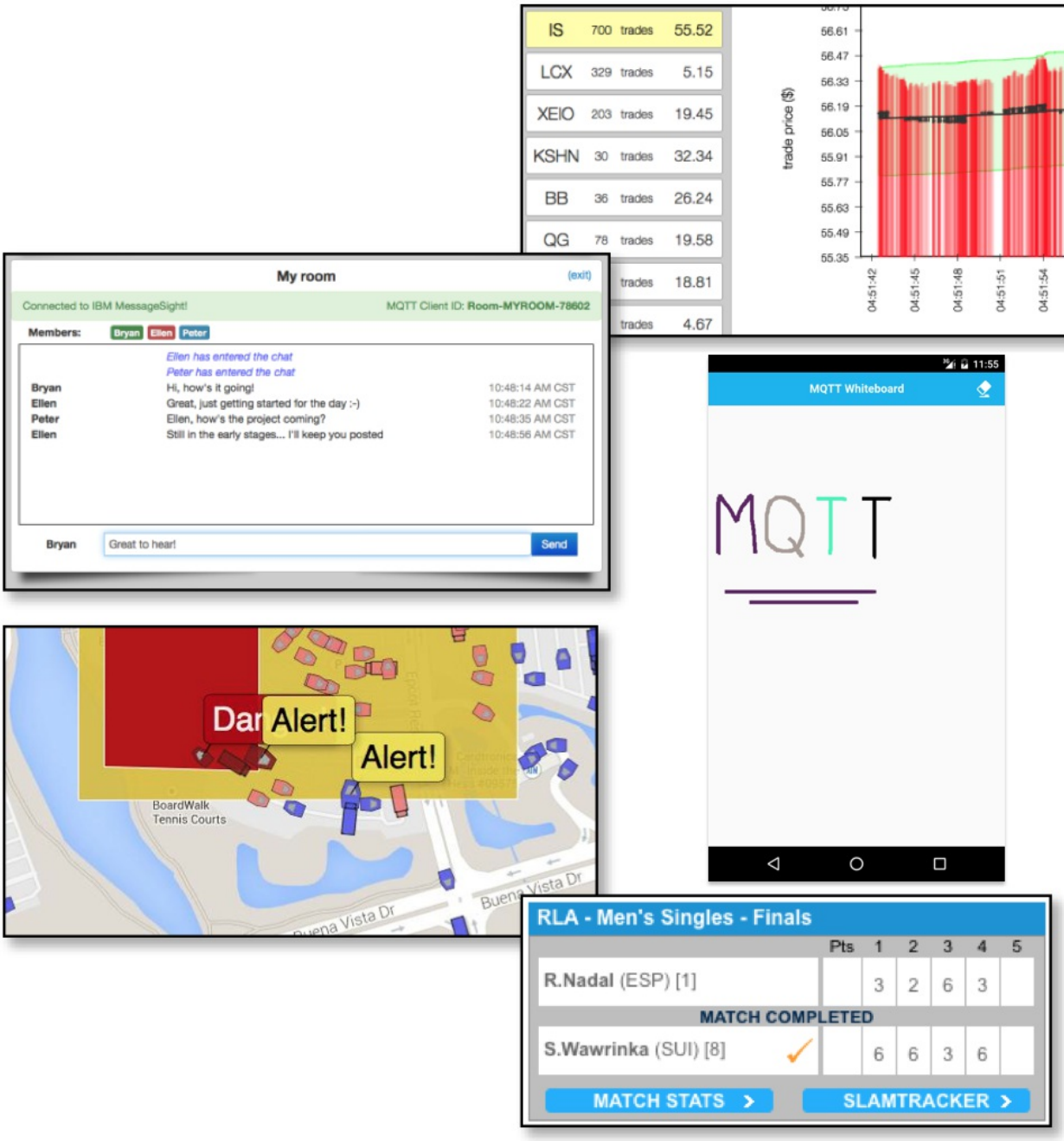
MQ Telemetry Transport provides:

- Reliable message delivery over unreliable connections
- Secure message delivery to the enterprise
- One-to-many message delivery (publish/subscribe)
- New real-time push of data from server (no polling)
- Minimal footprint on-the-wire (only 2-byte header)
- Reduced battery usage

MQ Telemetry Transport enables real-time data push from server to mobile devices, which makes it ideal for dynamic mobile applications.

Examples:

- Stock ticker
- Chat application
- Collaboration apps (Whiteboard)
- Real-time emergency alerts
- Live match score updates



MQTT client libraries are available in Java™, C, JavaScript™, C++, Python, Objective-C, and many other programming languages.

The Eclipse Paho project (http://www.eclipse.org/paho/ (http://www.eclipse.org/paho/)) provides many open source MQTT clients (http://www.eclipse.org/paho/ (http://www.eclipse.org/paho/)).

IBM provides MQTT clients in a Mobile Messaging & M2M Client Pack, available from the IBM Messaging community on developerWorks (https://developer.ibm.com/messaging/messagesight/).

This tutorial uses the Eclipse Paho JavaScript MQTT client for publish/subscribe messaging in MobileFirst Platform® application.

The MQTT clients have a simple API for publish/subscribe messaging.

The following examples use the Eclipse Paho JavaScript MQTT client to **Connect** to an MQTT broker, **Subscribe** to an MQTT topic, **Receive and process** messages from the broker, and **Publish** a message on an MQTT topic

Connect

```
function connect() {
  client = new Messaging.Client(hostname, port, clientId);
  client.onMessageArrived = onMsgCallback;
  client.onConnectionLost = onConnLostCallback;
  client.connect({ onSuccess: onSuccessCallback });
}
```

Publish

```
function publish(topic, data) {
  var msg = new Messaging.Message(data);
  msg.destinationName = topic;
  client.send(msg);
}
```

Subscribe

```
function subscribe(topic) {
  client.subscribe(topic);
}
```

Receive

```
function onMsgCallback(msg) {
  var topic = msg.destinationName;
  var data = msg.payloadString;
  console.log(topic, data);
}
```

Building a collaborative application (Whiteboard)

In this tutorial, you build a dynamic application (Whiteboard) with MobileFirst and MQ Telemetry Transport that lets users draw on a shared canvas in real time. Create this application in two steps:

- Build the MobileFirst application
  - Implement the "single-user" mode (run the sample without any MQTT publishes).
  - The application captures touch/click events, and you can paint on the canvas.
- Implement collaboration by using an MQTT client and server
  - Add a JavaScript MQTT client to the Whiteboard application that communicates with other clients through an MQTT server.
  - The MQTT server options are covered in a subsequent slide.

> **Note:** MQTT data does not use the MobileFirst Platform security and authentication mechanism.

The Whiteboard application uses MQTT messaging to provide a shared canvas for all users

Each Whiteboard publishes all the drawing actions as MQTT messages on a topic that is unique to the application session.

Each Whiteboard also subscribes to the set of all the drawing topics, and draws the actions of the others that are based on this data.

With MQTT publish/subscribe messaging capabilities, the real-time collaborative experience can scale to many connected applications.

This scenario cannot be efficiently implemented by using traditional polling (HTTP) or mobile push notifications.

- To provide a real-time experience, drawing actions should be reflected on other canvasses within milliseconds.
- HTTP polling is high bandwidth (each request requires a now client connection) and result in poor latency.
- Push notifications minimize bandwidth, but are inappropriate for small in-application updates.
- MQTT publish/subscribe messaging minimizes bandwidth and latency: an MQTT connections is established once from client to server, and messages are pushed directly to the application with low latency.

Whiteboard - Collaboration

To create a shared canvas, each Whiteboard client publishes draw, stop, and clear actions on a topic that is unique to the client. The type of action and options are described in the MQTT message payload as JSON data.

Example: **draw**

- Topic: `whiteboard/<clientId>`
- Payload: `{"type": "draw", "position": [105, 240], "color": "#F90000"}`

Example: **stop**

- Topic: `whiteboard/<clientId>`
- Payload: `{"type" : "stop", "color": "#F90000" }`

Example: **clear**

- Topic: `whiteboard/<clientId>`
- Payload: `{"type" : "clear" }`

When another Whiteboard client receives this message, `WhiteBoard.onMessage` is called.

Whiteboard - MQTT client

On application load, invoke connect the client:

main.js

```
var canvasContainer = document.querySelector('#canvas-container');

// TODO: update connection details, HOST and PORT
var wb = new WhiteBoard({
  host: 'YOUR_MQ_SERVER_HOST',
  port: 61623,
  container: canvasContainer
});

wb.connect();
```

Adding listeners for touch events, resize, and clear button

main.js

```
// ...

var canvas = wb.getCanvasElement();

canvas.addEventListener('touchstart', function (e) {
  var touch = e.touches[0];

  var x = touch.pageX;
  var y = touch.pageY - touch.target.offsetParent.offsetTop;

  wb.draw(x, y);
}, false);

canvas.addEventListener('touchend', function () {
  wb.stop();
}, false);

canvas.addEventListener('touchmove', function (e) {
  var touch = e.touches[0];

  var y = touch.pageY - touch.target.offsetParent.offsetTop;

  wb.draw(touch.pageX, y);
}, false);

window.addEventListener('resize', function () {
  wb.resize(canvasContainer.clientWidth, canvasContainer.clientHeight);
});

document.querySelector("#eraser").addEventListener('click', function () {
  wb.clear();
});
```

When you initialize and connect the MQTT client, specify the following configuration parameters and connection options.

MQTT Client Configuration	
host	The DNS name or IP of the MQTT broker host
port	The port number for the host
clientId	A unique MQTT identifier for this client (1 - 23 characters)
onMessageArrived	A callback that is called when a message is delivered to the client
onConnectionLost	A callback that is called when the connection is lost

Connection options	
keepAliveInterval	A period of inactivity (in seconds) after which the client disconnects
onSuccess	A callback that is called after a successful connection to the broker
onFailure	A callback that is called after an unsuccessful connection to the broker

If you do not have an MQTT broker available, see Connecting to an MQTT broker later in this tutorial for details about how to find your own broker.

whiteboard.js

```
(function(window){

  // ...

  function WhiteBoard(config) {
    // ...
    this.client = new Messaging.Client(config.host, config.port, this.uuid);
    // ...
  }
  // ...
  WhiteBoard.prototype.connect = function () {

    this.client.onMessageArrived = (function (self) {
      return function (message) {
        self.onMessage(message);
      };
    })(this);

    this.client.onConnectionLost = (function (self) {
      return self.onConnectionLost;
    })(this);

    var connectOptions = {};
    connectOptions.keepAliveInterval = 3600;
    connectOptions.onSuccess = (function (self) {
      return function () {
        self.onConnect();
      };
    })(this);
    connectOptions.onFailure = (function (self) {
      return self.onFailure;
    })(this);

    this.client.connect(connectOptions);
  };
  // ...
  window.WhiteBoard = WhiteBoard;
})(window);
```

Whiteboard - Adding callbacks

Next, implement the onMessage and onConnect callbacks.

When the MQTT connection succeeds, subscribe to the wildcard topic of all Whiteboard applications: `whiteboard/+`

You must make the difference between the draw and clear actions that come from other Whiteboards from the ones that your Whiteboard initiates.

For the actions that you initiate, publish an MQTT message. If you receive the action from an MQTT message, do not republish this action as a new message.

whiteboard.js

```
(function(window){
```

```
// ...
WhiteBoard.prototype.onConnect = function () {
    this.client.subscribe('whiteboard/+');
};

WhiteBoard.prototype.onFailure = function () {
    alert('Failed to connect!');
};

WhiteBoard.prototype.onConnectionLost = function () {
    alert('Connection lost!');
};

WhiteBoard.prototype.onMessage = function (message) {
    var topic = message.destinationName;
    var payload = message.payloadString;
    if (topic.indexOf("whiteboard/") == 0) {
        var sourceUUID = topic.split("/")[1];
        // don't process own actions
        if (sourceUUID == this.uuid) {
            return;
        }

        try {
            var data = JSON.parse(payload);

            if (data.type == "clear") {
                return this.clear(true);
            }

            var canvasId = colorToUUID(data.color);

            var canvasElement = document.querySelector('#' + canvasId);

            if (!canvasElement) {
                canvasElement = document.createElement('canvas');
                canvasElement.id = canvasId;
                canvasElement.width = this.container.scrollWidth;
                canvasElement.height = this.container.scrollHeight;


                this.container.appendChild(canvasElement);

                this.canvases[canvasId] = new Canvas(canvasElement, {
                    color: data.color,
                    width: 10
                });
            }

            var canvas = this.canvases[canvasId];

            if (data.type == 'draw') {
                draw(data.position[0], data.position[1], canvas, false, this);
            } else if (data.type == 'stop') {
                canvas.end();
            }
```

```
            } catch (e) {
                console.error(e);
            }
        }
    };

    // ...

    window.WhiteBoard = WhiteBoard;
})(window);
```

MQ Telemetry Transport provides three qualities of service to deliver messages between clients and servers:

- QoS 0 = "at most once"
- QoS 1 = "at least once"
- QoS 2 = "exactly once"

A retained message is retained on the MQTT server and transmitted to new subscribers to the message topic.

whiteboard.js

```
(function(window){

    // ...

    WhiteBoard.prototype.broadcastEvent = function (event) {
        var topic = 'whiteboard/' + this.uuid;

        if (typeof event.type === 'undefined') {
            throw {message: 'event type is required'};
        }

        var message = new Messaging.Message(JSON.stringify(event));
        message.destinationName = topic;
        message.qos = 0;
        message.retained = false;

        this.client.send(message);
    };

    window.WhiteBoard = WhiteBoard;
})(window);
```

Extending the Application

This collaboration scenario can be modified and extended to build other types of real-time applications:

- **Stock ticker:** clients subscribe to trade data for an exchange (for example, MarketData/NYSE/IBM) and update in real time.
- **Real-time emergency alerts:** clients subscribe to a topic for their own geographic location, and the emergency services publish alerts onto these same topics (for example, `emergency/<X>/<Y>/alert`).
- **Chat application:** clients publish and subscribe on a topic for a particular chat room (for example, chat/room123/user1).

Connecting to an MQTT broker

An MQTT server is required to broker messages between MQTT clients. To support JavaScript MQTT clients (such as Eclipse Paho), the MQTT server must implement the WebSocket transport.

Several MQTT servers are available, both open source and commercial.

- Development MQTT servers:
  - Mosquitto (open source, multi-platform)
  - IBM Message Sight for Developers (virtual appliance image)
- Enterprise MQTT servers:
  - IBM MessageSight (appliance)

Mosquitto

Mosquitto is an open source, lightweight implementation of MQ Telemetry Transport V3.1, part of the Eclipse Paho messaging project.

The Mosquitto broker is available for many different platforms (Windows, OS X, Linux distributions). For more information go to:

- http://mosquitto.org (http://mosquitto.org)
- http://projects.eclipse.org/projects/technology.mosquitto (http://projects.eclipse.org/projects/technology.mosquitto)

Download Mosquitto:
http://mosquitto.org/download/ (http://mosquitto.org/download/)

IBM MessageSight

IBM MessageSight for Developers is a virtual IBM MessageSight appliance image.

IBM MessageSight is a low-latency, reliable, and scalable messaging server with strong security and easy management.
The developer image can be run with virtualization software such as VMWare, Oracle VirtualBox, and KVM.

For more information go to:

- http://ibm.com/messagesight (http://ibm.com/messagesight)
- https://www.ibmdw.net/messaging/messagesight/ (https://www.ibmdw.net/messaging/messagesight/)
- http://www.youtube.com/watch?v=kcEDoRqhkhA (http://www.youtube.com/watch?v=kcEDoRqhkhA)

Sample application

The sample **MQTTWhiteboard** is a Corodova application that demonstrates use of MQTT.
Click to download (https://github.com/MobileFirst-Platform-Developer-Center/MQTTWhiteboard/tree/release80) the Cordova project.

Sample usage

1. Before running the application you need to update your MQTT broker host and port number in `main.js`. You also need to update your Content Security Policy (CSP) inside `index.html` to include your MQTT broker.
2. From a **Command-line** window, ensure the sample is registered in the MobileFirst Server by running the command: `mfpdev app register`.
3. Add a platform by running the `cordova platform add` command.
4. Run the Cordova application by running the `cordova run` command.