Sparkplug en un coup d'œil

une spécification pour les données MQTT

Source : Adobe Stock

Tam Hanna (Hongrie)

Le fameux protocole MQTT est l'un des outils les plus simples pour connecter des appareils électroniques distribués, tels que des capteurs, des contrôleurs et des appareils de stockage de données. Tant que les messages transmis respectent les règles de la norme et que le courtier peut être adressé correctement, l'échange de données est possible. Un nouveau protocole appelé Sparkplug tente de formaliser les données de l'utilisateur avec une couche supplémentaire, ce qui facilite l'offre de services auxiliaires et la création d'un écosystème diversifié, entre autres avantages. Voici un premier coup d'œil - avec quelques exemples pratiques.

Le protocole MQTT [1], basé sur TCP/IP, est devenu la norme de l'internet des objets ; il est léger, polyvalent et facile à comprendre. En outre, MQTT peut très bien gérer les participants au réseau qui ne sont que temporairement actifs ou accessibles. Cependant, MQTT n'assure qu'une communication fiable - l'organisation des données de l'utilisateur est du ressort du développeur.

En plus, MQTT n'a offert pendant longtemps qu'une authentification rudimentaire, et la gestion des appareils connectés (au-delà de la transmission du message *Last Will* [2]) nécessitait une intervention du développeur ou l'utilisation d'un second service. L'absence de normalisation signifiait également que ce travail devait être effectué par chaque développeur et, dans le pire des cas, être refait pour chaque projet.

Le nouveau protocole, appelé Sparkplug, impose désormais un modèle aux données utilisateur transmises via MQTT, ce qui garantit enfin une structure organisée et gérable.

Dans cet article, nous examinerons les concepts de base et réaliserons les premiers essais pratiques avec cette technique, qui n'en est encore qu'à ses débuts.

Veuillez également lire le document du standard !

Pour des contraintes d'espace, cet article technique ne peut pas offrir une description exhaustive Sparkplug.

Néanmoins, la spécification du protocole dans sa version 3.0 est rigoureusement détaillée. Il est donc recommandé de conserver le document PDF de 140 pages [3] à proximité. Cet article inclut fréquemment des références croisées aux sections de ce document où des informations complémentaires sont disponibles.

Premier aperçu

Créée à l'origine par Cirrus Link, la norme Sparkplug [3] est gérée depuis quelque temps par la Fondation Eclipse, qui tente depuis longtemps de se positionner en tant que « touche-à-tout » dans le secteur de l'IdO.

La deuxième version du protocole a introduit l'utilisation des *Google Protocol Buffers* [4] comme format de données de conteneur. La version 3.0 de Sparkplug, exclusivement utilisée dans cet article, était axée sur la clarification des paramètres définis dans la spécification. Pour comprendre un système Sparkplug, examinons le schéma présenté dans la **figure 1**, qui illustre un réseau IdO.
Tout comme dans un réseau MQTT ordinaire, on trouve un serveur MQTT au centre. Il convient de noter qu'il s'agit - en général - d'un serveur MQTT ordinaire. Selon la spécification, les implémentations MQTT désignées comme *Sparkplug-Compliant MQTT Server* dans la version de base doivent seulement remplir les quatre critères suivants, qui ne sont pas particulièrement compliqués :

Un serveur MQTT conforme à Sparkplug DOIT prendre en charge
. . . la publication et la souscription sur QoS 0
. . . la publication et la souscription sur QoS 1
. . . tous les aspects des messages Will, y compris l'utilisation du drapeau retain et du QoS 1
. . . tous les aspects du drapeau *retain*

Les *Sparkplug-Aware MQTT Servers*, quant à eux, sont des implémentations avancées qui supportent les réseaux Sparkplug grâce à une « intelligence à valeur ajoutée » - leurs exigences exactes sont listées dans le document de spécification à la section 12.66.
Il convient de noter que la majorité des composants de l'infrastructure présentés dans la figure 1 ne sont pas absolument nécessaires pour un réseau MQTT - la version «de base» se compose uniquement d'un serveur MQTT et d'un *Sparkplug Edge Node*, qui peut être implémenté comme un client MQTT.
Le diagramme contient des éléments familiers - les *Edge Nodes* sont les capteurs ou les stations distantes qui transmettent au réseau les informations à traiter sous la forme de messages MQTT structurés selon les règles de Sparkplug.
Les clients MQTT qui reçoivent ces messages et les utilisent à des fins commerciales sont appelés « applications hôtes ». L'application hôte principale est utile dans la mesure où elle est particulièrement liée aux nœuds Edge individuels. La norme parle d'une application dont l'état en ligne ou hors ligne affecte le comportement du nœud Edge.
L'élément important suivant est la structure du message, décrite en

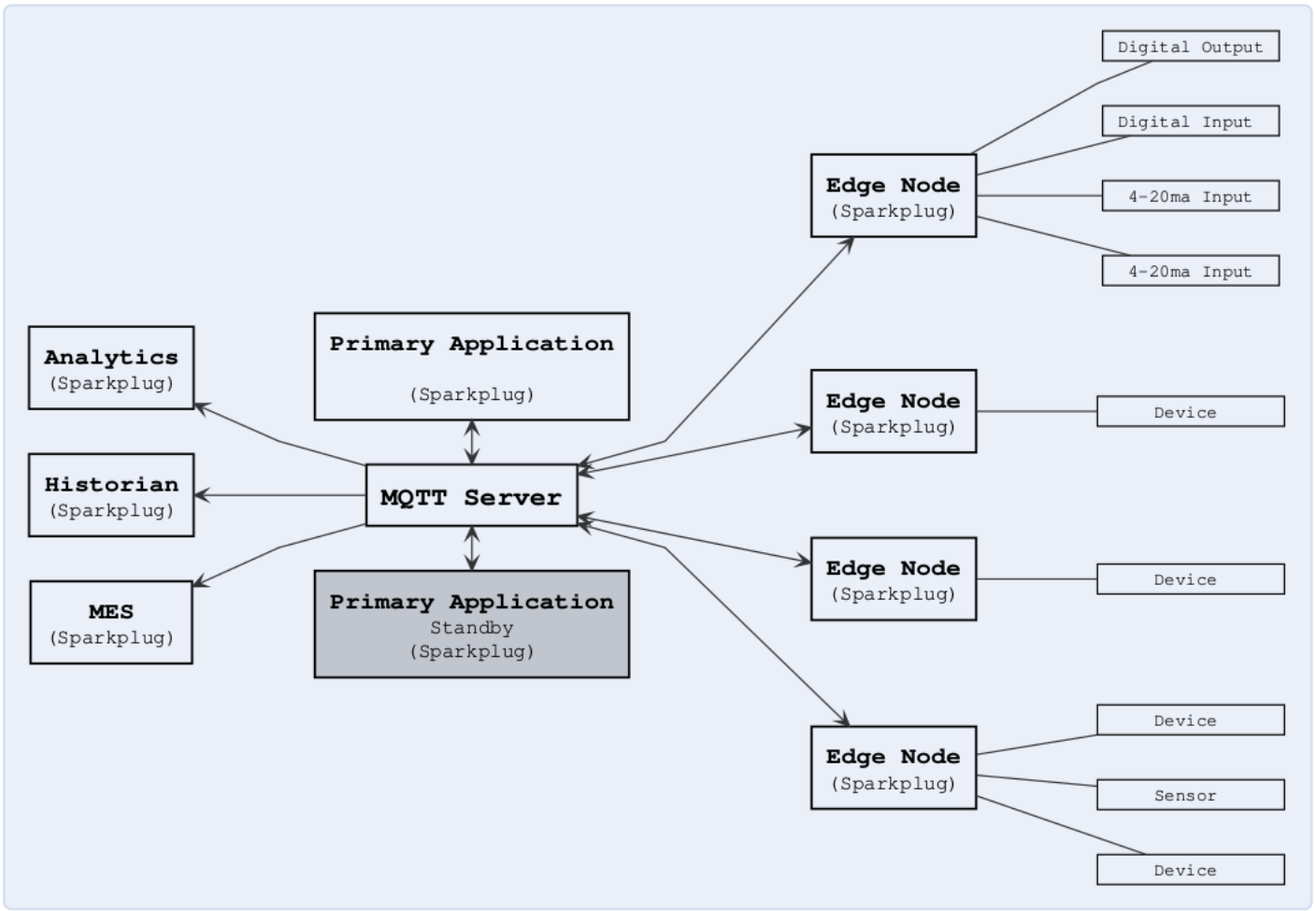

Figure 1. Ces composants fonctionnent ensemble pour créer un système Sparkplug. (Source : [3])

- **NBIRTH** – Birth certificate for Sparkplug Edge Nodes
- **NDEATH** – Death certificate for Sparkplug Edge Nodes
- **DBIRTH** – Birth certificate for Devices
- **DDEATH** – Death certificate for Devices
- **NDATA** – Edge Node data message
- **DDATA** – Device data message
- **NCMD** – Edge Node command message
- **DCMD** – Device command message
- **STATE** – Sparkplug Host Application state message

Figure 2. Il n'existe que neuf types de message. (Source : [3])

détail dans la section 4.1. Lorsqu'ils envoient des messages, les clients MQTT doivent respecter la structure suivante lors de la création de l'infrastructure du thème :

```
namespace/group_id/message_type/edge_node_id/[device_id]
```

`namespace` est utilisé pour «encoder» les canaux compatibles avec Sparkplug, et doit toujours être `spBv1.0`. `group_id` décrit la structure - analogue au *Tag* utilisé par divers fournisseurs de services cloud, il s'agit d'un moyen de spécifier plus en détail l'emplacement des nœuds individuels.
Le `message_type` est plus important. Au moment de la publication de cet article, la Fondation Eclipse n'en a spécifié que neuf, qui sont listés dans la **figure 2**.
Les attributs `edge_node_id/[device_id]` décrivent le dispositif terminal et le dispositif secondaire responsables de l'envoi du message. Il convient de noter que la norme MQTT permet de créer des « filtres » avec différents caractères spéciaux - il est donc possible, par exemple, d'enregistrer tous les messages de type particulier comme pertinents. De plus amples informations à ce sujet sont disponibles sur [5].

Analyse de la structure du message

Le deuxième facteur à prendre en compte lorsque l'on travaille avec la spécification Sparkplug est le rôle des différents messages, comme le montre la figure 2. Étant donné que la spécification décrit le flux de données avec des diagrammes de temps (déroutants), nous essaierons de regrouper les types de messages importants par rôle et par payload. Toutefois, si vous souhaitez consulter les diagrammes vous-même, la section 5 de la spécification est recommandée.
Commençons par les messages `DBIRTH` et `NBIRTH`. Comme le suggère le mot « BIRTH » (naissance) dans les noms, ces messages indiquent que de nouvelles parties du réseau Sparkplug deviennent disponibles. `NBIRTH` indique l'apparition d'un nouveau nœud, tandis que `DBIRTH` indique l'apparition d'un nouvel appareil.
Il est important que la charge utile des messages fournisse une structure qui permette au récepteur de générer une image numérique complète (souvent appelée « jumeau numérique ») de tout ce qui change au niveau du point d'extrémité qui vient d'être ajouté. Il découle de la même logique que `NDEATH` et `DDEATH` sont tous deux responsables de la disparition des points d'extrémité.
Les messages `NDEATH` sont responsables de la disparition d'un nœud et sont envoyés par le serveur MQTT lorsque le nœud disparaît. L'enregistrement fait partie du message `NBIRTH`. Le message `DDEATH`, responsable de la disparition d'un appareil, est envoyé par le nœud - il est important de noter que l'envoi de ce message ne dépend que du nœud.
Les messages `NDATA` et `DDATA` permettent de transmettre des valeurs calculées et d'envoyer des commandes afin de modifier l'état d'un attribut détenu dans le dispositif terminal pour lui donner une nouvelle valeur.
Le message `DDATA` est important parce qu'il introduit le concept de rapport par exception (*Report by Exception*) - dans la spécification, l'abréviation RBE est souvent utilisée pour cette procédure. Dans le monde de sparkplug cela signifie que les changements d'état ne doivent être envoyés à l'hôte que si une « exception » - un changement qui mérite l'attention - s'est produite.
La procédure normalement utilisée - désignée dans la spécification sous le nom de *Time-Based Reporting* - est également autorisée, mais elle est explicitement décrite comme indésirable :

"Again, time-based reporting can be used instead of RBE, but is discouraged and typically unnecessary."

Payload : Google Protocol Buffers

Ceux qui sont familiers avec la programmation bas niveau connaissent les problèmes liés à la sérialisation des structures de données. Avec les Protocol Buffers, Google propose une normalisation - conceptuellement basée sur JSON et autres - qui facilite la création de « conteneurs sérialisables indépendants de la plateforme ».
La spécification Sparkplug s'appuie sur les Protocol Buffers comme charge utile, probablement aussi en raison de leur prise en charge très étendue, pour laquelle il existe maintenant des bibliothèques adaptées dans presque tous les langages de programmation. Les développeurs sérieusement intéressés par Sparkplug peuvent consulter le site web de Protocol Buffers [4].
Il convient de noter que les Protocol Buffers représentent un protocole binaire. Dans la pratique, ainsi que dans le document de spécification, vous voyez souvent des annotations JSON structurées comme suit :

```
{
 «timestamp»: <timestamp>,
 «metrics»: [{
  «name»: <metric_name>,
  «alias»: <alias>,
  «timestamp»: <timestamp>,
  «dataType»: <datatype>,
  «value»: <value>
 }],
 «seq»: <sequence_number>
}
```

Il s'agit d'un format redéfini à partir des données binaires, qui n'a rien à voir avec les informations envoyées physiquement sur les ondes. En principe, l'extrait présenté ici nous montre tout ce que l'on peut attendre d'un message Sparkplug.
`timestamp`, utilisé depuis l'époque Unix, doit être indiqué en UTC et est un entier de 64 bits qui décrit les millisecondes écoulées. Le champ `metrics`, qui est ici « à valeur unique », contient les données

de l'utilisateur à transmettre dans le message.
Enfin, il existe un numéro de séquence qui, comme dans d'autres protocoles, permet de garantir l'intégrité de la transmission des données. Une explication complète des champs de données transmis dans les différents messages dépasserait le cadre de cet article et serait peu utile - si vous souhaitez réaliser une « implémentation de bas niveau », il est recommandé de consulter la section 6 de la spécification. Dans la pratique, les bibliothèques sont généralement utilisées pour effectuer l'implémentation.

Passons à la pratique !

Dans l'intérêt de garder la spécification « concrète », le document officiel mentionne le fait qu'un Raspberry Pi utilisant une carte d'E/S fait plus ou moins « officiellement » partie de la spécification et sert d'exemple d'implémentation d'un nœud Edge Sparkplug. Dans ce contexte, il est intéressant de savoir ce que nous voulons utiliser comme point d'entrée des données ou comme implémentation de l'application hôte (sur un PC).
Il convient de noter que la Fondation Eclipse propose une liste d'implémentations compatibles [6]. Pour être listé, un produit doit passer un test de compatibilité - mais cela dépasse le cadre de cet article. Eclipse Tahu [7] fournit également une bibliothèque *wrapper* presque clés en main qui facilite le déploiement des paquets.
Dans les étapes suivantes, nous voulons utiliser *Ignition* d'Inductive Automation - c'est l'une des rares implémentations d'un hôte Sparkplug.
Un courtier de messages est nécessaire en arrière-plan - l'auteur utilise Ubuntu 20.04 LTS, c'est pourquoi nous allons démarrer Mosquitto dans un conteneur Docker.
Un fichier de configuration est nécessaire ici - Mosquitto a sauvegardé la configuration dans la version 2.0.0, c'est pourquoi la connexion de clients anonymes n'est pas autorisée sans la modification montrée ici :

```
tamhan@TAMHAN18:~$ cat mosquitto.conf
allow_anonymous true
listener 1883
persistence true
persistence_location /mosquitto/data/
log_dest file /mosquitto/log/mosquitto.log
```

Le démarrage proprement dit se déroule ensuite comme suit :

```
tamhan@TAMHAN18:~$ docker run -it
 -p 1883:1883 -p 9001:9001
 -v $(pwd)/mosquitto.conf:/mosquitto/config/
  mosquitto.conf eclipse-mosquitto
```

Après avoir configuré le courtier MQTT, l'étape suivante consiste à visiter le site web [8], où nous téléchargeons Ignition. Le logiciel est livré avec un assistant d'installation, qui est activé comme suit :

```
tamhan@TAMHAN18:~/Downloads$ chmod +x
 ignition-8.1.25-linux-64-installer.run
tamhan@TAMHAN18:~/Downloads$
 ./ignition-8.1.25-linux-64-installer.run
```

Il convient de noter que cette commande ne démarre pas toujours le conteneur Mosquitto, alors que le système s'intègre plus tard dans le processus de démarrage de *systemd*. Si un comportement inhabituel se produit parce que le serveur Mosquitto n'a été démarré qu'après le conteneur de la plate-forme, il est possible d'y remédier avec la séquence de commandes suivante :

```
tamhan@TAMHAN18:/usr/local/bin/ignition$
 ./ignition.sh stop
tamhan@TAMHAN18:/usr/local/bin/ignition$
 ./ignition.sh start
```

Lors de l'installation, le mot de passe du superutilisateur est demandé afin d'effectuer l'« intégration ». Dans les étapes suivantes, l'auteur a décidé d'utiliser le répertoire d'installation */usr/local/bin/ignition* - ceci est important à savoir car le script requis ultérieurement pour la désinstallation y est caché.
Dans la zone de sélection des modules pour Ignition, vous devez d'abord sélectionner l'option *Custom* afin de marquer toutes les options dans la fenêtre de liste des modules qui apparaît ensuite. Il est primordial de sélectionner, entre autres, les paquets *Web Browser* et *Web Dev*.
L'assistant d'installation propose ensuite de démarrer la plate-forme. Cela fonctionne normalement et est confirmé par un message de succès structuré comme suit :

```
INFO [IgnitionInstaller       ]
 [2023/03/12 22:15:50]:
 Gateway Address: http://localhost:8088
```

Dans certains cas, le système se bloque. Vous devez alors vous rendre dans le répertoire d'installation pour forcer un redémarrage manuel :

```
tamhan@TAMHAN18:/usr/local/bin/ignition$ ./ignition.
sh start
```

L'étape suivante consiste à visiter l'URL d'Ignition Gateway et à installer l'édition *Maker*. Il s'agit d'une version légèrement limitée du produit, qui est gratuite pour les utilisateurs non commerciaux.
Après le démarrage réussi (le message Ignition `Gateway is starting` pourrait apparaître à l'écran pendant un certain temps), nous sélectionnons l'option *Enable QuickStart* pour mettre la plate-forme dans un état « quick-start ready ».
Le système est entièrement modulaire en interne. C'est pourquoi, dans la première étape, nous optons pour l'option *Config Systems Modules* pour activer la gestion des modules. Apparaît alors un groupe d'archives prêtes à l'emploi [9] avec des modules de fournisseurs tiers - pour utiliser la plateforme Sparkplug, nous avons besoin des éléments *MQTT Transmission* et *MQTT Engine*, que nous installons un par un depuis l'interface web.
À l'étape suivante, nous passons à l'option de configuration *Config Mqttengine MQTT Engine Settings*, où nous sommes informés - comme le montre la **figure 3** - que l'instance Mosquitto située dans le conteneur Docker a été trouvée avec succès.
Il existe également un autre paramètre sous *Config Mqtttransmission*

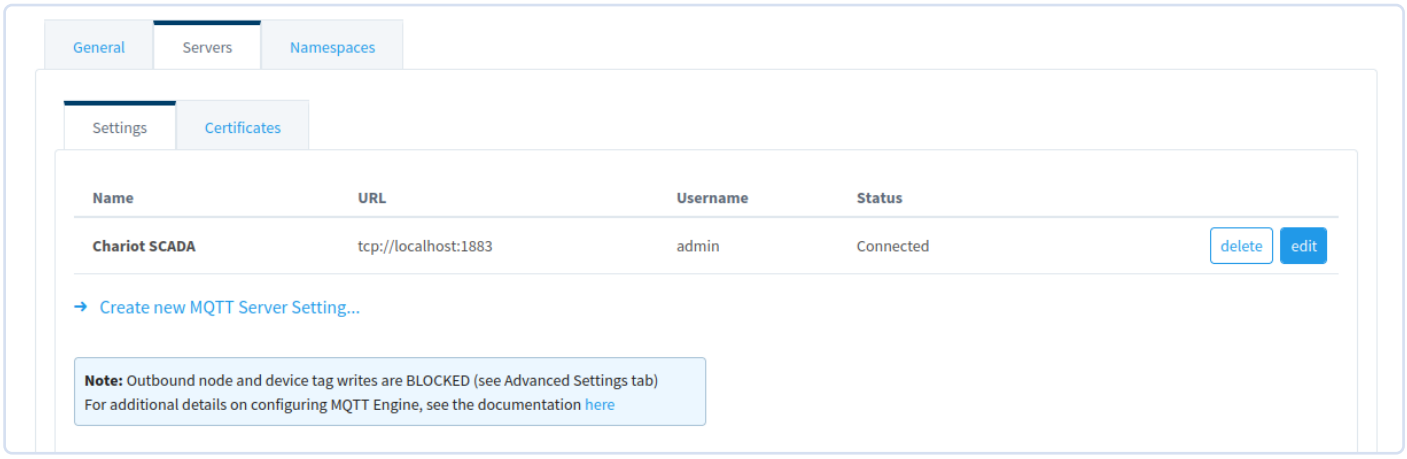

Figure 3. L'instance Mosquitto a été automatiquement détectée.

MQTT Transmission Settings, dont l'exactitude doit également être vérifiée. Pour que la plate-forme du serveur reste compacte, le système est également conçu de manière modulaire. Le bouton Get Designer vous permet de télécharger un composant séparé appelé *Designer*, qui vous permet de configurer les paramètres existants dans la plate-forme. Téléchargez l'archive dans la première étape et décompressez-la - le démarrage effectif de l'application se fait alors comme avec n'importe quel autre utilitaire de ligne de commande :

```
tamhan@TAMHAN18:~/designerlauncher/app$
 ./designerlauncher.sh
```

L'étape suivante consiste à établir un contact avec l'application *Designer* récemment lancée. Pour ce faire, nous choisissons d'abord l'option *Add Designer*, puis nous cliquons sur l'option *Localhost* pour ajouter une nouvelle « *platform entry* ». Le bouton *Open Designer* nous permet ensuite de modifier les informations.
Nous choisissons ensuite l'option *SampleQuickStart* et cliquons sur *Open* pour charger l'exemple de démarrage. En cliquant sur l'option

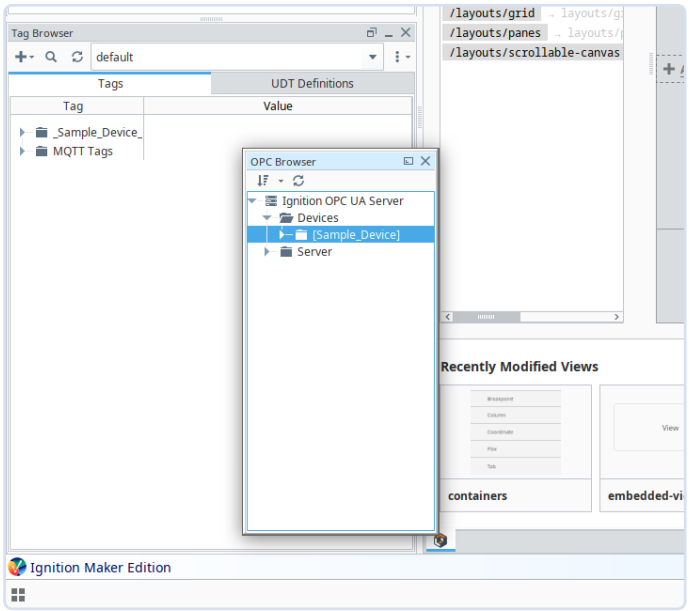

Figure 4. La fonction « glisser-déposer » dans la section « par défaut » est utile.

View Panels OPC Browser, un autre panneau s'ouvre, dans lequel différentes sources de données OPC sont disponibles. Développez la section *Expand Devices* pour afficher l'appareil [`Sample_Device`]. Une opération de glisser-déposer est alors nécessaire, comme le montre la **figure 4**.
Ensuite, nous devons veiller à fermer le Designer et à utiliser l'option *Save* (enregistrer) et *Close* (fermer) pour enregistrer et « charger » toutes les modifications apportées dans l'application locale. Nous pouvons alors revenir à l'interface web, où nous pouvons définir une nouvelle configuration en utilisant l'option *MQTT TRANSMISSION Settings Transmitters* Create new Settings. Il est alors important que le texte « default » soit saisi dans le champ *Tag Provider* - l'enregistrement des modifications met à jour le statut du système.
Ensuite, nous ouvrons la console de la station de travail, où nous activons le test *subscriber* contenu dans le paquet Mosquitto et le connectons au canal `spBv1.0/#`. La seule chose importante à savoir ici est que le symbole # sert de symbole de remplacement, généralement représenté par `*` :

```
tamhan@TAMHAN18:~$ mosquitto_sub
 -h localhost -t spBv1.0/#
```

Nos efforts sont récompensés par l'apparition de fichiers binaires, comme le montre la **figure 5**.

Configuration du Raspberry Pi

L'étape suivante nécessite un ordinateur de traitement - puisque la spécification fait référence au Raspberry Pi à plusieurs reprises, nous voulons l'utiliser dans les étapes suivantes. L'auteur utilise un Raspberry Pi 4 avec une version assez récente du système d'exploitation. La séquence de commandes suivante est nécessaire pour activer les composants :

```
pi@raspberrypi:~/sparkspace $ git clone
 https://github.com/eclipse/tahu.git
Cloning into 'tahu'...
. . .
pi@raspberrypi:~/sparkspace
 $ cd tahu/java/examples/raspberry_pi
pi@raspberrypi:~/sparkspace/tahu/java/
 examples/raspberry_pi $ mvn clean install
```

Le code d'exemple fourni par la Fondation Eclipse n'est pas fonctionnel

en soi, mais doit être complété par les dépendances et les bibliothèques nécessaires en utilisant la gestion des paquets Maven dans la première étape.
L'étape suivante concerne le fichier *SparkplugRaspberryPiExample.java*, que l'on peut ouvrir dans un éditeur comme suit :

```
pi@raspberrypi:~/sparkspace/tahu/java/
 examples/raspberry_pi/src/main/java/
 org/eclipse/tahu $ pico SparkplugRaspberryPiExample.java
```

L'exemple de code suppose qu'un courtier MQTT externe sera responsable du traitement effectif des informations. Dans la configuration des postes de travail de l'auteur, son adresse IP est *192.168.1.68*, c'est pourquoi une adaptation est nécessaire :

```
public class SparkplugRaspberryPiExample
implements MqttCallbackExtended {
 private static final String
  DFLT_MQTT_SERVER_HOST_NAME = «192.168.1.68»;
```

Ensuite, Maven doit être rappelé pour télécharger les éléments manquants :

```
pi@raspberrypi:~/sparkspace/tahu/java/
 examples/raspberry_pi $ ls
 dependency-reduced-pom.xml  pom.xml
 src  target  THIRD-PARTY.txt
pi@raspberrypi:~/sparkspace/tahu/java/
 examples/raspberry_pi $ mvn clean install
```

Si vous avez déjà installé une version de Java sur votre Raspberry Pi, vous pouvez lancer le programme résultant comme suit :

```
pi@raspberrypi:~/sparkspace/tahu/java/
 examples/raspberry_pi/target $ java
 -jar example_raspberry_pi-1.0.1.jar
```

Dans la plupart des cas, une erreur se produit à ce stade, liée à la bibliothèque *libdio* - ce n'est pas critique pour nous, car nous nous pencherons sur la structure du programme à l'étape suivante et nous voulons examiner de plus près certains aspects intéressants.
Tout d'abord, nous examinerons le calcul du numéro de séquence, qui s'effectue de la manière suivante :

```
// Used to get the sequence number
private long getNextSeqNum() {
 long retSeq = seq;
 if (seq == 256) {
  seq = 0;
 } else {
  seq++;
 }
 return retSeq;
}
```

Il est également intéressant d'examiner le format dans lequel les informations fournies par la carte d'E/S (qui n'est pas utilisée ici) sont présentées pour être interprétables par l'hôte. Le code nécessaire à cet effet ressemble à ceci :

```
SparkplugBPayload payload =
 new SparkplugBPayloadBuilder(getNextSeqNum())
  .setTimestamp(new Date())
// Create an «Inputs» folder of process variables
```

Figure 5. Les caractères spéciaux indiquent qu'il s'agit de données binaires.

```
  .addMetric(new MetricBuilder
    (PibrellaInputPins.A.getPin().getDescription(),
     MetricDataType.Boolean,
     pibrella.getInput(PibrellaInputPins.A)
      .isHigh()).createMetric())
  .addMetric(new MetricBuilder
    (PibrellaInputPins.B.getPin().getDescription(),
     MetricDataType.Boolean,
      pibrella.getInput(PibrellaInputPins.B).
       isHigh()).createMetric())
. . .
.createPayload();

// Publish the Device BIRTH Certificate now
executor.execute(
new Publisher(NAMESPACE + «/» +
     groupId + «/DBIRTH/» +
      edgeNode + «/» + deviceId, payload));
```

Les messages sont alors généralement reçus de la même manière en utilisant l'API Target :

```
public void messageArrived
   (String topic, MqttMessage message) throws Exception {
 System.out.println(«Message Arrived on topic « + topic);

// Initialize the outbound payload if required.
 SparkplugBPayloadBuilder outboundPayloadBuilder =
  new SparkplugBPayloadBuilder
     (getNextSeqNum()).setTimestamp(new Date());

String[] splitTopic = topic.split(«/»);
if (splitTopic[0].equals(NAMESPACE) &&
  splitTopic[1].equals(groupId) &&
  splitTopic[2].equals(«NCMD») &&
  splitTopic[3].equals(edgeNode)) {

 SparkplugBPayload inboundPayload =
  new SparkplugBPayloadDecoder().
  buildFromByteArray(message.getPayload());
...
```

Dans ce contexte, il est également intéressant de noter que le standard est capable de traiter des charges utiles étendues. Un bon exemple en est la boucle `for` suivante, qui énumère les différentes commandes fournies sur le Raspberry Pi :

```
 SparkplugBPayload inboundPayload =
  new SparkplugBPayloadDecoder().
  buildFromByteArray(message.getPayload());

for (Metric metric : inboundPayload.getMetrics()) {
 System.out.println(«Metric: « +
      metric.getName() + « :: « +
      metric.getValue());
if (metric.getName().equals
   («Node Control/Next Server»)) {
 System.out.println(«Received a Next Server command.»);
} else if (metric.getName().
     equals(«Node Control/Rebirth»)) {
 publishBirth();
} else if (metric.getName().
     equals(«Node Control/Reboot»)) {
 System.out.println(«Received a Reboot command.»);
} else if (metric.getName().
     equals(«Node Control/Scan Rate ms»)) {
 scanRateMs = (Integer) metric.getValue();
if (scanRateMs < 100) {
// Limit Scan Rate to a minimum of 100ms
 scanRateMs = 100;
}
```

Pour les étapes suivantes, cependant, nous voulons utiliser un exemple un peu plus simple, qui est disponible sous `/home/pi/sparkspace/tahu/java/examples/simple/src/main/java/org/eclipse/tahu`.

Ouvrez d'abord le fichier Java et ajustez deux des dizaines de variables membres :

```
public class SparkplugExample
    implements MqttCallbackExtended {

// HW/SW versions
. . .
 private String serverUrl =
  «tcp://192.168.1.68:1883»;
 private long PUBLISH_PERIOD = 1000;
  // Publish period in milliseconds
```

Comme précédemment, vous devez bien entendu adapter la valeur stockée dans la variable `serverUrl` à la situation de votre réseau. La réduction du délai `PUBLISH_PERIOD` permet alors une transmission plus rapide des données à l'application principale.
Après avoir enregistré le fichier modifié, il est nécessaire de retourner dans le répertoire racine du projet et de lancer une nouvelle compilation avec l'outil de gestion de paquets Maven :

```
pi@raspberrypi:~/sparkspace/tahu/java/
    examples/simple $ mvn clean install
```

Le passage au dossier racine est principalement nécessaire parce que le fichier *pom.xml* s'y trouve - il est chargé de contrôler la compilation générale du projet.
La récompense de nos efforts est un fichier *.jar*, qui se trouve dans le répertoire *~/sparkspace/tahu/java/examples/simple/target*. L'activation se fait ensuite :

```
pi@raspberrypi:~/sparkspace/tahu/java/
   examples/simple/target $ java
```

```
-jar example_simple-1.0.1.jar
```

Pour voir les résultats, vous devez revenir à l'application Designer, où le résultat s'affiche comme le montre la **figure 6**.

Pour les oublieux : Réinitialisation de la passerelle

Soyons honnêtes : il est facile d'oublier le mot de passe de la passerelle. Heureusement, ce problème n'est pas difficile à résoudre. Allez dans le répertoire d'installation et exécutez les trois commandes suivantes :

```
tamhan@TAMHAN18:/usr/local/bin/ignition$
    ./gwcmd.sh --passwd
Password has been reset. Gateway needs to be restarted.
tamhan@TAMHAN18:/usr/local/bin/
    ignition$ ./ignition.sh stop
Stopping Ignition-Gateway...
tamhan@TAMHAN18:/usr/local/bin/
   ignition$ ./ignition.sh start
Starting Ignition-Gateway with systemd...
Waiting for Ignition-Gateway...
running: PID:382296
```

Après avoir redémarré la passerelle, vous pouvez vous connecter via l'URL - la passerelle présente alors au premier utilisateur une fenêtre dans laquelle vous pouvez spécifier un nouveau nom d'utilisateur et le mot de passe correspondant pour le compte de l'administrateur.

Un système efficace

Avec Sparkplug, la Fondation Eclipse entre dans la course avec un écosystème solide qui semble bien adapté pour « dominer » la prolifération incontrôlée que l'on peut sans doute trouver dans le domaine du MQTT. Cependant, comme dans le cas de nombreux autres systèmes, la véritable valeur d'un tel standard ne devient apparente que lorsqu'il est largement adopté. L'effet de réseau est incontournable.

230038-04

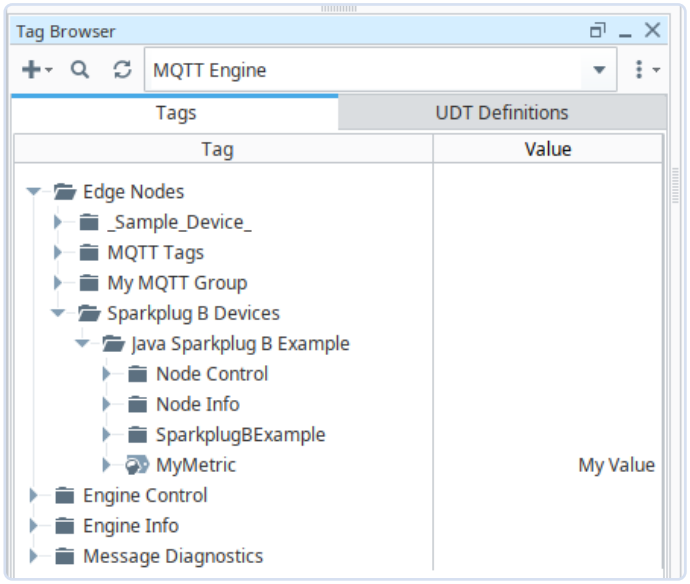

Figure 6. Assurez-vous que la boîte combo est correctement paramétrée.

Questions ou commentaires ?

Envoyez un courriel à l'auteur (tamhan@tamoggemon.com), ou contactez Elektor (redaction @elektor.fr)

À propos de l'auteur

Avec plus de 20 ans d'expérience, Tam Hanna est un ingénieur spécialisé dans l'électronique, l'informatique et les logiciels. Il est designer indépendant, auteur de plusieurs ouvrages et journaliste (@tam.hanna sur Instagram). Durant ses moments libres, il se consacre à la conception et à la production de solutions imprimées en 3D. Il nourrit également une passion pour le commerce et la dégustation de cigares haut de gamme.

LIENS

[1] MQTT basics: https://hivemq.com/mqtt
[2] MQTT: Last Will: https://hivemq.com/blog/mqtt-essentials-part-9-last-will-and-testament
[3] Standard document : https://sparkplug.eclipse.org/specification/version/3.0/documents/sparkplug-specification-3.0.0.pdf
[4] Google Protocol Buffers : https://developers.google.com/protocol-buffers
[5] MQTT: Topics : https://hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices
[6] Logiciel Sparkplug : https://sparkplug.eclipse.org/compatibility/compatible-software
[7] Eclipse Tahu : https://github.com/eclipse/tahu
[8] Inductive Automation Ignition download: https://inductiveautomation.com/downloads
[9] Modules tiers : https://inductiveautomation.com/downloads/third-party-modules/8.0.17