

UNIVERSITÉ DU QUÉBEC EN OUTAOUAIS

INF4173 - Projet Synthèse

Rapport Final

Présenté à M. Michal Iglewski et M. Kamel Adi

Par

Vincent Crispin Emond & Kossi Ahlin Cornelus Madjri

4/20/2017

Table des matières

1. Introduction	4
2. Projet iSerre.....	4
2.1 Description	4
2.2 Objectifs	4
2.3 Motivations	5
2.4 Architecture.....	5
2.4.1 Capteurs	6
2.4.2 Actionneurs	6
2.4.3 Sink	7
2.4.4 Passerelle.....	7
2.4.5 MQTT/MQTT-SN.....	8
2.4.6 iSerre Sensor Network (iSN).....	9
2.4.7 MQTT Broker	9
2.4.8 OpenHab	10
3. Projet iSerre Sensor Network (iSN)	10
3.1 Description	10
3.2 Objectifs	11
3.2.1 Objectifs initiaux.....	11
3.2.2 Évolution des objectifs	11
3.2.3 Objectifs finaux.....	12
3.3 Motivations	13
3.4 Architecture du protocole	13
3.4.1 Client/Serveur	14
3.4.2 Spécification des trames	14
3.4.3 Paramètres de configuration et constantes.....	19
3.4.4 Procédure de connexion	22
3.4.5 Procédure de keepalive.....	24
3.4.6 États-transitions	25

3.4.7	Calcul de la mesure du côté serveur	32
3.5	Notre implémentation d'iSN	36
3.5.1	Structure des projets.....	36
3.5.2	Classes et fonctions importantes	43
3.5.3	Interopérabilité	53
3.5.4	Procédure de débogage	57
3.5.5	Choses à améliorer	63
4.	Apprentissages	64
4.1	Programmation embarquée C/C++	64
4.2	MQTT/MQTT-SN.....	64
4.3	Kinetis design studio	64
4.4	XCTU	64
4.5	Branchements	65
5.	Défis et difficultés rencontrées	66
5.1	Apprentissage.....	66
5.2	Codage avec ressources limitées	66
5.3	Problèmes de connexion entre la passerelle et le broker	66
6.	Conclusion	67
	Glossaire.....	68
	Annexe.....	69

1. Introduction

Les sciences naturelles sont un domaine d'étude très important dans notre société. Elles permettent entre autre de préserver nos écosystèmes en cas de catastrophes ou de changements climatiques, de déterminer les conditions idéales à la culture de plantes pour l'agriculture ou le domaine médical et plus encore.

Pour aider à la recherche dans ce domaine il serait bien de pouvoir contrôler l'environnement dans lequel les plantes sont cultivées.

Notre projet offre une solution à ce problème sous la forme d'une serre électronique qui offre à l'utilisateur un contrôle total sur ses paramètres environnementaux. Que ce soit la température, la luminosité, l'humidité etc. Le nom choisi pour cette serre est iSerre.

Dans ce rapport, nous allons d'abord expliquer en quoi consiste le projet iSerre dans son ensemble. Ensuite, nous allons nous concentrer sur le sujet de notre projet synthèse qui consistait à faire la conception et l'implémentation d'un protocole de communication pour le réseau de capteurs à l'intérieur de la serre. En troisième lieu nous allons faire une synthèse des apprentissages acquis durant le projet, suivi d'une synthèse des défis et difficultés rencontrées pour finalement terminer avec la conclusion.

2. Projet iSerre

2.1 Description

Il s'agit ici de la conception une serre entièrement automatisée contenant plusieurs capteurs prenant des relevés environnementaux qui seront affichés sur une interface utilisateur

2.2 Objectifs

Le projet **iSerre** vise à mettre en place une serre automatisée

- Les paramètres environnementaux (température, humidité etc.) sont affichés en temps réel sur une interface web.
- L'utilisateur peut modifier ces paramètres à l'aide d'actionneurs (plaque chauffante, soupape, etc.) contrôlables via l'interface web.
- Un historique des données est disponible pour fin de recherche, analyse ou consultation.

2.3 Motivations

La serre se veut un outil de recherche indispensable. Notamment, pour le département des sciences naturelles de l'UQO :

- Étudier les effets du changement climatique sur les plantes.
- Déterminer les paramètres optimaux pour la culture de différentes espèces de plantes.
- S'occuper des plantes pendant des périodes d'absence prolongées.

2.4 Architecture

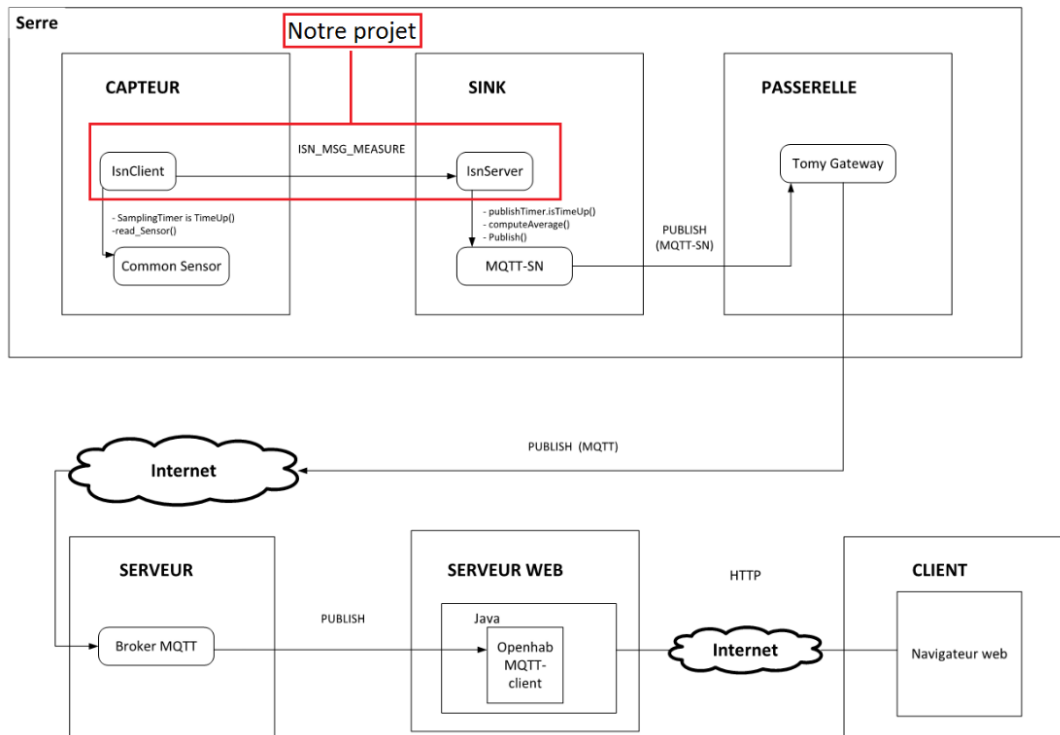


Figure 1 Architecture d'iSerre

2.4.1 Capteurs

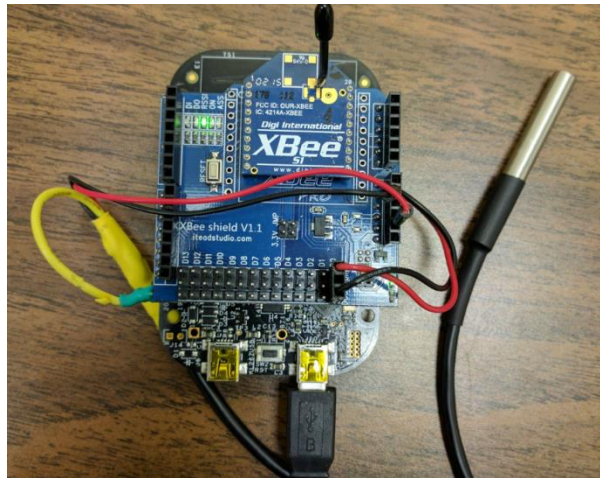


Figure 2 Capteur de température

Ce module sera chargé de faire l'étude et les expérimentations des capteurs. Notamment les connexions avec la carte Freescale FRDM-KL26 ainsi que les tests de fonctionnement avant et après déploiement dans la serre.

Dans le cadre de notre projet le capteur sur lequel nous avons porté notre attention est celui mesurant la température « Waterproof DS18B20 »

2.4.2 Actionneurs

Il s'agit des périphériques qui permettent d'agir sur les paramètres environnementaux dans la serre. Il y a une plaque chauffante pour augmenter la température, une électrovanne pour l'humidité du sol, un ventilateur pour changer le taux d'oxygène ou de dioxyde de carbone dans l'air et un humidificateur pour changer le taux d'humidité dans l'air. Il y a aussi un relai qui est un dispositif qui permet de contrôler tous ces circuits.

2.4.3 Sink

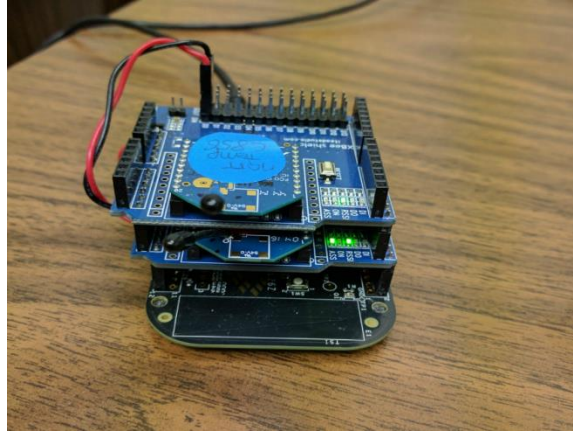


Figure 3 Sink

Le « Sink » est un point de ralliement pour tous les différents type capteurs (température, humidité, actionneurs, etc.). Il agit comme un nœud de synchronisation d'informations. Non seulement le sink s'occupe du traitement des données recueillies par les différents capteurs mais il se charge aussi de transmettre les données traitées au Gateway. Les protocoles utilisés entre ce niveau et le prochain sont MQTT-SN/DigiMesh. Le Sink est implémenté sur une carte Freescale FRDM KL26Z.

2.4.4 Passerelle

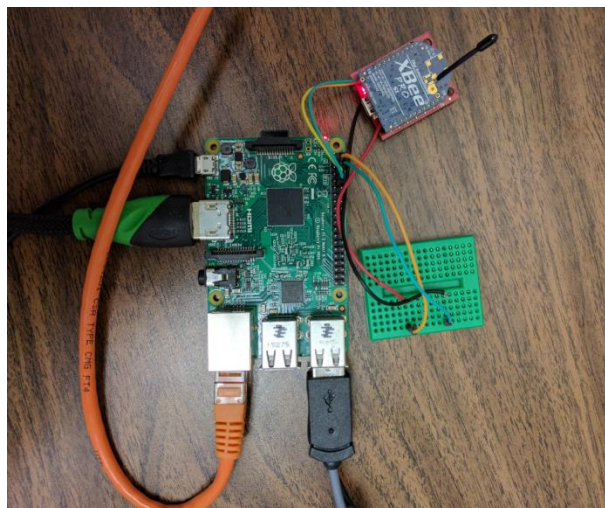


Figure 4 Passerelle

La passerelle implémentée sur une Raspberry Pi (RPI) fait la conversion entre MQTT-SN et MQTT. Cette passerelle reçoit des trames MQTT-SN à travers le réseau Xbee et les transforme en requêtes MQTT vers le serveur (broker) à travers le réseau TCP/IP. Le code de la passerelle est un code open source appelé TomyGateway.

2.4.5 MQTT/MQTT-SN

MQTT (MQ Telemetry Transport) est un protocole léger de messagerie « publish-subscribe » basé sur le protocole TCP/IP. Il a été initialement développé par Andy Stanford-Clark (IBM) et Arlen Nipper (EuroTech), puis offert à la communauté Open Source. MQTT v3.1.1 est maintenant un standard OASIS, la spécification du protocole est disponible en HTML et PDF.

On distingue dans ce protocole des clients et le broker/serveur. Le broker contient et met à jour des topics auxquels les clients peuvent s'abonner. Les clients peuvent ainsi s'abonner ou publier des messages sur un topic. Tout client abonné à un topic reçoit tous les messages que d'autres clients publient sur ce topic.

Il existe de nombreux brokers MQTT disponibles, ils varient dans leurs fonctionnalités et certains d'entre eux mettent en œuvre des fonctionnalités additionnelles.

Les principaux brokers open-sources sont :

- ActiveMQ
- JoramMQ, OW2 JORAM
- Mosquitto
- RabbitMQ

Le broker utilisé dans le cadre de ce projet est le broker open-source appelé « mosquitto ».

MQTT-SN est conçu pour être aussi proche que possible de MQTT, mais est adapté aux particularités d'un environnement de communication sans fil, comme une faible bande passante, des pannes de liaison élevées, une longueur de message courte, etc. Il est également optimisé pour la mise en œuvre sur Dispositifs à faible coût et à piles avec des ressources de traitement et de stockage limitées.

MQTT-SN (MQTT pour Sensor Networks) est une version plus légère de MQTT, implémenté pour une bonne communication dans les réseaux de capteurs. Ces capteurs sont typiquement très limités au niveau de leurs ressources.

Le MQTT-SN est spécifiquement pour les réseaux de capteurs et ne dépend pas de TCP-IP pour qu'il fonctionne. Il peut fonctionner sur n'importe quelle couche de transport, comme ZigBee.

Il existe trois types de composants MQTT-SN :

- clients MQTT-SN,
MQTT-SNgateways (GW) et
- MQTT-SNforwarders.

Les clients MQTT-SN se connectent à un serveur MQTT via un MQTT-SN GW à l'aide du protocole MQTT-SN. Un MQTT-SN GW peut ou non être intégré à un serveur MQTT. Dans le cas d'un GW autonome, le protocole MQTT est utilisé entre le serveur MQTT et le MQTT-SN GW. Sa fonction principale est la traduction entre MQTT et MQTT-SN. Les clients MQTT-SN peuvent également accéder à un GW via un redirecteur au cas où le GW n'est pas directement connecté à leur réseau. Dans le sens opposé, il décapsule les images qu'il reçoit de la passerelle et les envoie aux clients, inchangé aussi.

2.4.6 iSerre Sensor Network (iSN)

iSerre Sensor Network ou iSN est un protocole de communication utilisé pour la communication entre les capteurs/actionneurs et les sinks. Il a été conçu et implémenter spécifiquement pour le projet iSerre et est le sujet de notre projet.

Auparavant, le protocole iSN a été partiellement implémenté afin de faire fonctionner toute la serre à temps pour une démonstration. Seules les trames de commande et de mesure sont implémentées et le fonctionnement actuel est le suivant :

- Toutes les adresses des modules Xbee sont hardcodées (sinks, capteurs, actionneurs).
- Les capteurs envoient en permanence (trame iSN_Mesure) la valeur mesurée au sink correspondant).
- Le sink fait la moyenne de toutes les mesures qu'il a reçues et publie sur le topic correspondant.
- Le sink des actionneurs est abonné aux différents topics 'actionneur'. Lorsqu'un message est publié sur un des topics, il envoie une trame iSN_Command à l'actionneur correspondant et celui-ci s'active ou se désactive.

2.4.7 MQTT Broker

Le broker contient et met à jour des topics auxquels les clients peuvent s'abonner. Les clients peuvent ainsi s'abonner ou publier des messages sur un topic. Tout client abonné à un topic reçoit tous les messages que d'autres clients publient sur ce topic. Le broker utilisé dans le cadre de ce projet est le broker open-source appelé « mosquitto » mais nous avons aussi notre propre broker qui roule sur un serveur à l'UQO.

2.4.8 OpenHab

OpenHAB est un logiciel permettant d'intégrer différents systèmes et technologies de domotique dans une seule solution qui permet des règles d'automatisation étendues et qui offre des interfaces utilisateur uniformes. Via l'intermédiaire d'OpenHab, nous serons en mesure d'afficher en temps réel toutes les informations recueillis par les différents capteurs présents dans la serre sur une interface utilisateur conviviale. Il sera possible d'ajuster les paramètres des différents capteurs via la même interface utilisateur.

Ce logiciel open source basé sur java offre un haut degré d'abstraction pour la conception d'applications domotiques. Il a été conçu pour fonctionner sur une variété de systèmes d'exploitation, ce qui explique sa flexibilité.

3. Projet iSerre Sensor Network (iSN)

3.1 Description

Notre projet synthèse vise à faire la conception et l'implémentation d'un protocole de communication entre capteurs (actionneurs) et sink. Ce protocole est nommé iSerre Sensor Network. Nous allons, à partir de maintenant, abréger le nom à iSN à des fins de simplicité.

Dans le cas de la communication capteur-sink le but premier du protocole est d'acheminer les mesures des capteurs vers leur sink respectif.

Dans le cas de la communication actionneur-sink le but premier du protocole est d'envoyer des commandes du sink à l'actionneur.

Dans tous les cas le protocole définit une procédure de connexion automatique des capteurs/actionneurs ainsi qu'une procédure de « keepalive » pour une gestion saine des connexions au niveau du serveur.

Dans cette section nous allons d'abord donner une description du protocole indépendante de son implémentation. Pour être plus précis, certaines choses sont spécifiques à l'implémentation d'iSN et dépasse la portée de la définition du protocole.

Par exemple, notre implémentation doit prendre en compte l'interopérabilité entre iSN et le protocole MQTT-SN parce que notre environnement utilise MQTT-SN pour le transport de données à la passerelle. En revanche, iSN se préoccupe seulement du transport des mesures entre le capteur et le sink. La gestion des données après ce point est laissée à la discrétion du développeur.

Après avoir défini la portée du protocole nous allons expliquer les détails spécifiques à notre implémentation d'iSN.

3.2 Objectifs

Dans cette section nous allons décrire les objectifs de notre projet synthèse ainsi que les objectifs qui ont guidé la conception et l'implémentation du protocole iSN.

Il faut noter que nous avons commencé le projet avec des objectifs initiaux qui sont différents des objectifs finaux du projet.

Au fur et à mesure que le projet a avancé, la nature du travail à faire est devenu plus clair pour nous ce qui a mené à une évolution des objectifs. Certains objectifs ont apparus, aussi la méthodologie pour atteindre certains objectifs a aussi changé.

3.2.1 Objectifs initiaux

Les objectifs initiaux étaient :

- Améliorer l'efficacité et la stabilité de l'architecture de communication de la serre.
- Implémenter (coder) le protocole iSerre Sensor Network (iSN) sur les cartes microcontrôleurs.
- Pouvoir ajouter ou retirer de nouveaux capteurs dans la serre sans avoir à les configurer manuellement.
- Obtenir de l'expérience de programmation embarquée avec C++ et de programmation de protocoles de communication.

3.2.2 Évolution des objectifs

Les objectifs et la méthodologie que nous utilisons pour les atteindre ont évolué au cours du projet. Dans cette section nous allons expliquer cette évolution.

3.2.2.1 Objectifs

- « Améliorer l'efficacité et la stabilité de l'architecture de communication de la serre. » est un objectif très général. Nous avons maintenant des objectifs spécifiques à la conception et l'implémentation d'iSN. Ces objectifs seront définis dans la prochaine section.
- « Implémenter (coder) le protocole iSerre Sensor Network (iSN) sur les cartes microcontrôleurs. » cet objectif ne s'arrête pas seulement à l'implémentation d'iSN mais aussi au travail de conception du protocole.

3.2.2.2 Méthodologie

Dans notre plan de travail nous avons écrit :

« Nous allons vérifier si une implémentation du protocole de communication ZigBee qui est un standard de l'industrie pourrait être envisageable à la place du protocole propriétaire DigiMesh. Cela pourrait améliorer l'interopérabilité des composantes utilisées dans la serre. »

Après analyse du code nous avons découvert que le code pour MQTT-SN et le prototype d'iSN est profondément dépendant du format des trames en mode API spécifiques au protocole propriétaire DigiMesh. Changer pour le protocole ZigBee demanderait une réécriture significative du code et serait un projet en soit. Nous avons donc abandonné cette idée.

Un autre point qu'on avait mentionné était :

« Nous allons tenter de voir s'il est possible d'utiliser un seul module Xbee par sink au lieu de deux. »

Nous avons commencé à décrire une procédure pour utiliser qu'un seul Xbee dans notre rapport de progrès. Cette procédure proposait l'ajout de 16 bits à l'en-tête de toutes les trames iSN servant à identifier le protocole. Toutefois, nous avons abandonné cette option car iSN deviendrait trop « bruyant » et nous voulions garder un protocole léger.

3.2.3 Objectifs finaux

Les objectifs finaux se divisent en deux catégories. Il y a les objectifs qui se rapportent au projet synthèse en tant que tel et il y a les objectifs qui sont spécifiques au protocole iSN.

3.2.3.1 Objectifs du projet synthèse

- Améliorer l'efficacité et la stabilité de l'architecture de communication de la serre.
- Concevoir et implémenter le protocole iSerre Sensor Network (iSN) sur les cartes microcontrôleurs freescale.
- Pouvoir ajouter ou retirer de nouveaux capteurs dans la serre sans avoir à les configurer manuellement.
- Obtenir de l'expérience de programmation embarquée avec C++ et de programmation de protocoles de communication.

3.2.3.2 Objectifs spécifiques à iSN

- Le protocole doit être robuste : il doit fonctionner pendant de très longues périodes sans nécessiter une intervention humaine.
- Le protocole doit être générique : on doit pouvoir intégrer de nouveaux types de capteurs avec un minimum de changements au code.
- Le protocole permet une configuration : on doit pouvoir changer les paramètres des capteurs sans les redémarrer.
- Le capteur doit être capable de négocier son entrée dans le réseau automatiquement.
- Le protocole doit décrire une procédure permettant d'envoyer les mesures des capteurs à la passerelle.

3.3 Motivations

Plusieurs problèmes avec l'architecture initiale de la serre ont menés à la conception et à l'implémentation d'iSN :

- On doit recompiler le code de tous les capteurs si on change une composante comme un module radio Xbee.

```
/*-----  
*           Xbee module address  
*-----*/  
NWAddress64 tempActuator(0x0013A200, 0x4103DD61);  
NWAddress64 ledActuator (0x0013A200, 0x40C1C448);
```

Figure 5 Les adresses statiques.

Comme on peut voir sur la Figure 5 les adresses des modules Xbee des actionneurs étaient codées de manière statique dans le code du sink. Si on change un module Xbee alors on devait recompiler le code du sink en changeant ces deux lignes.

- Impossible d'ajouter un sink de rechange au réseau en cas de bris de matériel car les capteurs se connectent à des adresses fixes.

Pour la même raison que le premier point, on ne peut pas avoir d'équipement de rechange car il faudrait recompiler le code avec l'adresse du sink de rechange.

- Ajouter un nouveau type de capteur au réseau exige une réécriture totale du code.

Initialement, tout le code d'envoi de mesures au sink était inclus dans le code du capteur sans aucune modularisation. Si on voulait ajouter un nouveau type de capteur alors on devait réécrire tout ce code réseau dans le code du nouveau capteur ou au moins copier, coller et adapter le code au nouveau capteur.

- Pour changer un paramètre de configuration sur un capteur on doit recompiler son code.

Les paramètres de configuration du capteur et du sink étaient fixes dans le code. Soit sous forme de littéraux soit sous forme d'instructions préprocesseur « #define ». Il était donc impossible de changer les paramètres des capteurs sans les débrancher et les recompiler.

3.4 Architecture du protocole

Dans cette section, il sera question des détails spécifiques à iSN. Le lecteur peut considérer cette section comme une description formelle du protocole iSN sur laquelle il peut se baser pour l'implémenter. Les détails spécifiques à notre implémentation seront expliqués dans la section 3.5.

3.4.1 Client/Serveur

Le protocole iSN fonctionne sous un modèle client-serveur. Le client est implémenté du côté du capteur ou de l'actionneur tandis que le serveur est implémenté du côté du sink. Le protocole est orienté connexion, c'est-à-dire qu'une connexion doit être établie au préalable avant d'initier les communications. La procédure de connexion est définie à la section 3.4.4.

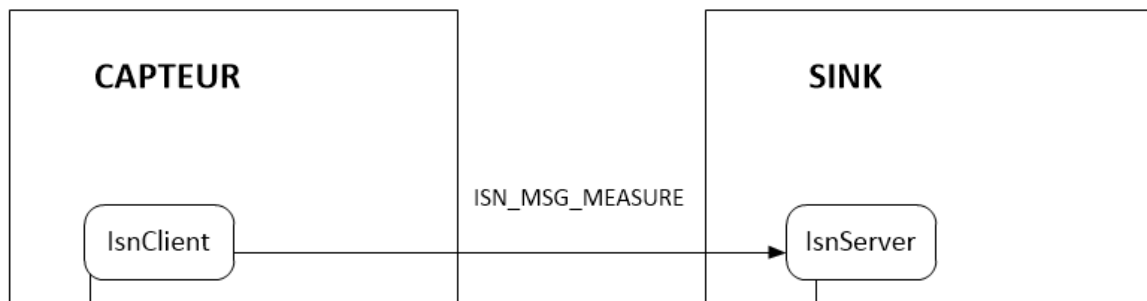


Figure 6 Architecture client-serveur

3.4.1.1 IsnClient

On appelle IsnClient l'implémentation du protocole qui réside du côté du client, c'est-à-dire, les capteurs ou les actionneurs. Cette partie de l'implémentation est responsable de l'initiation de la connexion et de l'envoi de mesure dans le cas des capteurs. Dans le cas des actionneurs, elle est responsable de la réception des commandes par le sink. La Figure 6 montre la position de IsnClient dans l'architecture. Pour le reste de la section 3 les mots : client et IsnClient sont équivalents, ces mots sous-entendent aussi capteur et actionneur. Lorsqu'on veut faire une distinction claire entre capteur et actionneur nous utilisons les mots capteur ou actionneur respectivement.

3.4.1.2 IsnServeur

On appelle IsnServeur l'implémentation du protocole qui réside du côté du serveur, c'est-à-dire, sur un nœud de type sink. Cette partie de l'implémentation est responsable de récolter les mesures de ses clients et d'écouter pour d'éventuelles connexions. Elle a aussi la responsabilité de gérer sa liste de clients en retirant les clients qu'elle juge mort et envoie aussi des commandes aux actionneurs dans le cas d'un nœud sink actionneur. La Figure 6 montre la position d'IsnServeur dans l'architecture. Pour le reste de la section 3 les mots IsnServeur, serveur et sink sont équivalents.

3.4.2 Spécification des trames

Le client et le serveur communiquent en envoyant des messages sous forme de trames. Le protocole iSN a été conçu avec la légèreté et simplicité en tête, cela se reflète dans le format des trames qui est très simple.

Format général des trames

Type de message (1 octet)	Données du message (variable)
---------------------------	-------------------------------

Figure 7 Format général d'une trame iSN

Toutes les trames dans le protocole iSN adoptent cette forme. Le premier champ d'un octet est obligatoire et est un identifiant pour le type de trame. Le deuxième champ a une longueur variable et n'est pas présent dans toutes les trames. La longueur de ce champ dépend du type de trame lue.

ISN_MSG_SEARCH_SINK

Type de message	Type d'équipement
0x01	1 octet

Type de message (1 octet) : Cette valeur est 0x01 pour les trames ISN_MSG_SEARCH_SINK.

Type d'équipement (1 octet) : Cette valeur indique le type de sink qu'on veut rechercher. Par exemple : Un sink température ou un sink humidité etc. Voir la section 3.4.3 pour les options disponibles.

Description : Cette trame est envoyée en diffusion par un IsnClient non connecté pour faire la découverte de sink dans le but d'établir une connexion.

ISN_MSG_SEARCH_SINK_ACK

Type de message
0x05

Type de message (1 octet) : Cette valeur est 0x05 pour les trames ISN_MSG_SEARCH_SINK_ACK.

Description : Cette trame est un accusé de réception envoyé d'un IsnServer en réponse à une trame ISN_MSG_SEARCH_SINK envoyée par un IsnClient. iSN ne définit pas de champ pour les adresses réseaux, un IsnServer ou IsnClient doit récupérer les adresses en utilisant les données du ou des protocoles qui encapsulent les trames iSN.

ISN_MSG_CONFIG

Type de message	Nombre	Nom du paramètre	Valeur du paramètre
0x02	1 octet	1 octet	2 octets

Type de message (1 octet) : Cette valeur est 0x02 pour les trames ISN_MSG_CONFIG.

Nombre (1 octet) : Un entier qui indique le nombre de couple nom valeur de 3 octets qui suivent. Par exemple : si cette valeur est 2 alors les champs nom du paramètre et valeur du paramètre se répètent 2 fois il faut donc lire 6 octets après le champ nombre.

Nom du paramètre (1 octet) : Ce champ contient une valeur qui identifie le paramètre de configuration qu'on veut changer. Voir la section 3.4.3 pour la liste des valeurs définies par le protocole, l'ajout de nouveaux paramètres est spécifique à l'implémentation et est laissé à la discrétion du développeur. Ce champ peut se répéter plusieurs fois selon la valeur du champ nombre mais il est toujours suivi d'un champ valeur du paramètre.

Valeur du paramètre (2 octets) : Ce champ contient la valeur du paramètre identifié par le champ nom du paramètre précédent. Ce champ peut se répéter plusieurs fois selon la valeur du champ nombre mais il sera toujours précédé d'un champ nom du paramètre.

Description : Cette trame est envoyée d'un IsnServer à tous ses IsnClients pour régler leurs paramètres de configuration elle peut être envoyée à tout moment. Elle est aussi envoyée une fois lors de la connexion d'un IsnClient à un IsnServeur afin de régler les paramètres initiaux. Lors de la connexion cette trame est seulement envoyée au client qui demande la connexion et non pas à tous les clients dans la liste de connexion du IsnServer. Elle sert aussi d'accusé de réception à la trame ISN_MSG_CONNECT.

ISN_MSG_CONFIG_ACK

Type de message
0x06

Type de message (1 octet) : Cette valeur est 0x06 pour les trames de type ISN_MSG_CONFIG_ACK.

Description : Cette trame est envoyée d'un IsnClient à un IsnServeur pour accuser réception d'une trame ISN_MSG_CONFIG.

ISN_MSG_COMMAND

Type de message	Commande
0x03	1 octet

Type de message (1 octet) : Cette valeur est 0x03 pour les trames ISN_MSG_COMMAND.

Commande (1 octet) : Un code de commande pour l'actionneur.

Description : Ces trames sont envoyées du sink à un actionneur pour lui faire effectuer une action. Pour l'instant aucune commande n'est définie par le protocole et l'ajout de commandes est spécifique à l'implémentation.

ISN_MSG_MEASURE

Type de message	Mesure
0x04	4 octets

Type de message (1 octet) : Cette valeur est 0x04 pour les trames ISN_MSG_MEASURE.

Mesure (4 octets) : Ce champ contient la mesure du capteur sous forme de float.

Description : Ces trames sont utilisées pour envoyer une mesure prise par un IsnClient capteur à un IsnServeur.

ISN_MSG_CONNECT

Type de message
0x07

Type de message (1 octet) : Cette valeur est 0x07 pour les trames ISN_MSG_CONNECT.

Description : Ce type de trame est envoyé d'un IsnClient à un IsnServeur pour initier une procédure de connexion.

ISN_MSG_NOT_CONNECTED

Type de message
0x08

Type de message (1 octet) : Cette valeur est 0x08 pour les trames ISN_MSG_NOT_CONNECTED.

Description : Ce type de trame est envoyé d'un IsnServeur à un IsnClient en réponse à un ISN_MSG_PING dans le cas où le IsnClient qui est à l'origine du ping ne figure pas dans la liste de clients du IsnServeur.

ISN_MSG_PING

Type de message
0x09

Type de message (1 octet) : Cette valeur est 0x09 pour les trames ISN_MSG_PING.

Description : Ce type de trame est envoyé d'un IsnClient à son IsnServeur pour vérifier que celui-ci est toujours actif.

ISN_MSG_PING_ACK

Type de message
0x0A

Type de message (1 octet) : Cette valeur est 0x0A pour les trames ISN_MSG_PING_ACK.

Description : Ce type de trame est envoyé d'un IsnServeur à un IsnClient en réponse à un ISN_MSG_PING dans le cas où le IsnClient qui est à l'origine du ping figure bel et bien dans la liste de clients du IsnServeur.

3.4.3 Paramètres de configuration et constantes

Dans cette section nous allons énumérer les constantes et paramètres de configuration définis par le protocole iSN.

3.4.3.1 Constantes

Le tableau suivant définit les constantes pour le type d'équipement. Les nœuds de type capteur ont des valeurs comprises entre 0x01 et 0x7F tandis que ceux de type actionneurs ont des valeurs entre 0x80 et 0xFF. Cette liste est sujette à extension dans de futures versions du protocole.

Nom de la constante	Valeur de la constante	Description
ISN_SENSOR_TEMP	0x01	Identificateur pour les nœuds sink et capteur de température.
ISN_SENSOR_HUMI	0x02	Identificateur pour les nœuds sink et capteur d'humidité du sol.
ISN_SENSOR_O2	0x03	Identificateur pour les nœuds sink et capteur d'oxygène.
ISN_SENSOR_CO2	0x04	Identificateur pour les nœuds sink et capteur de dioxyde de carbone.
ISN_SENSOR_HUMI_AIR	0x05	Identificateur pour les nœuds sink et capteur d'humidité de l'air.
ISN_ACTI_TEMP	0x81	Identificateur pour les nœuds sink et actionneur de type température (plaque chauffante).
ISN_ACTI_HUMI	0x82	Identificateur pour les nœuds sink et actionneur de type humidité de l'air. (Humidificateur)
ISN_ACTI_FAN	0x83	Identificateur pour les nœuds sink et actionneur de type ventilateur.
ISN_ACTI_VALVE	0x84	Identificateur pour les nœuds sink et actionneur de type humidité du sol (Électrovanne).
ISN_ACTI_RELAY	0x85	Identificateur pour les nœuds sink et actionneur de type relai. Le relai est un dispositif utilisé pour contrôler l'ouverture et la fermeture des différents circuits dans la serre.

3.4.3.2 Configuration à l'exécution

Le tableau suivant présente les noms des paramètres de configuration qui peuvent être envoyés d'un IsnServeur à un IsnClient via une trame ISN_MSG_CONFIG afin de modifier les paramètres ce celui-ci au moment de l'exécution.

Le nom du paramètre est une étiquette qu'on donne au paramètre pour l'identifier. Identificateur est l'entier qui est envoyé avec une trame ISN_MSG_CONFIG pour indiquer le type de paramètre à modifier. Valeur est la valeur du paramètre qui suit immédiatement l'identificateur dans la trame ISN_MSG_CONFIG.

Nom du paramètre	Identificateur	Description	Valeur
ISN_CONFIG_SAMPLING	0x01	Le délai après lequel IsnClient échantillonne la mesure de son capteur et l'envoi à son sink via une trame ISN_MSG_MEASURE.	Une valeur de type unsigned char. Le délai en secondes.
ISN_CONFIG_SAMPLING_DELAY	0x02	Pour éviter les collisions entre les trames de mesure. Un délai aléatoire est ajouté avant l'envoi d'une trame ISN_MSG_MEASURE. Ce paramètre indique le délai maximum qui peut être généré aléatoirement.	Une valeur de type unsigned char. Le délai maximum pouvant être généré en secondes.

3.4.3.3 Configuration à la compilation

Le tableau suivant définit les paramètres de configuration statiques du protocole. C'est-à-dire ceux qui sont déterminés au moment de la compilation.

Certaines trames sont accompagnées d'un accusé de réception. Si l'accusé n'est pas reçu dans un délai prescrit on dit que le « timeout » est atteint. Si le « timeout » est atteint l'émetteur essaye de renvoyer la trame pour un nombre déterminé de fois.

Dans le tableau suivant, « nom de la trame » indique le nom de la trame pour laquelle on attend un accusé de réception, « paramètre de timeout » indique l'étiquette qu'on donne au paramètre décrivant le nombre de secondes pendant lequel on attend l'accusé de réception et « paramètre nombre d'essais » indique l'étiquette qu'on donne au paramètre indiquant le nombre de fois

qu'on renvoie la trame dans le cas où l'accusé de réception n'est pas reçu dans le délai spécifié par le « timeout ».

Les valeurs des paramètres de « timeout » et « nombre d'essais » sont spécifique à l'implémentation du protocole. Dans notre implémentation « timeout » est en spécifié en secondes et « nombre d'essais » est un entier.

Nom de la trame	Paramètre de timeout	Paramètre nombre d'essais
ISN_MSG_SEARCH_SINK	ISN_CLIENT_SEARCH_TIMEOUT	ISN_CLIENT_SEARCH_RETRY
ISN_MSG_CONNECT	ISN_CLIENT_CONNECT_TIMEOUT	ISN_CLIENT_CONNECT_RETRY
ISN_MSG_PING	ISN_CLIENT_PING_TIMEOUT	ISN_CLIENT_PING_RETRY
ISM_MSG_CONFIG	ISN_SERVER_CONFIG_TIMEOUT	ISN_SERVER_CONFIG_RETRY

Le tableau suivant définit deux paramètres utilisés lors de la réception des trames ISN_MSG_MEASURE par le IsnServer. Les unités et les type de données de ces paramètres sont spécifique à l'implémentation.

Nom du paramètre	Description
ISN_SERVER_CONFIG_GRACE_DELAY	Lorsqu'une application externe demande la moyenne des mesures au IsnServer. Celui-ci vérifie d'abord si toutes les mesures de ses clients sont à jour. Si ce n'est pas le cas il attend pendant un délai spécifié par ce paramètre afin de donner une chance aux clients tardifs d'envoyer leurs valeurs.
ISN_SERVER_CONFIG_MISS_LIMIT	Le client qui échoue de fournir une mesure à jour pour le nombre de fois spécifiée par ce paramètre sera automatiquement déconnecté de son IsnServer.

Le paramètre suivant est utilisé par le client pour régler la procédure de keepalive.

Nom du paramètre	Description
ISN_CLIENT_CONFIG_KEEP_ALIVE	Lorsque la période de temps définie par ce paramètre est écoulée le client doit envoyer une trame ISN_MSG_PING à son serveur pour vérifier que la connexion tient toujours.

3.4.4 Procédure de connexion

Comme spécifié dans les objectifs du protocole, un IsnClient doit être capable d'automatiquement négocier son entrée dans le réseau. La Figure 8 illustre cette procédure de connexion automatique.

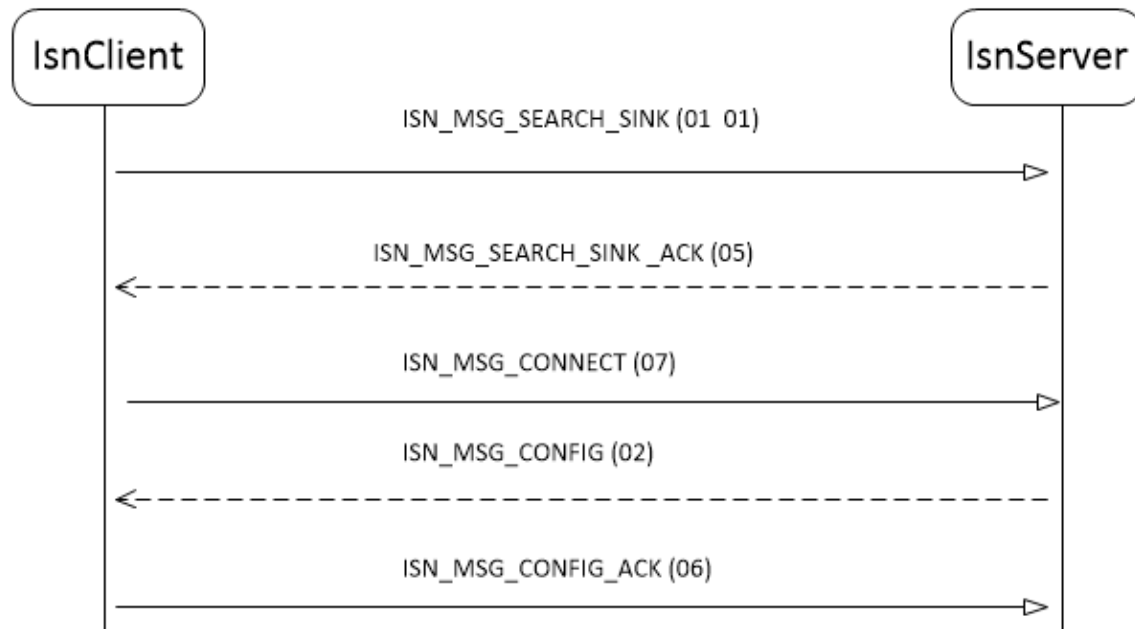


Figure 8 Procédure de connexion

Sur ce diagramme la direction des flèches indique le sens de la communication. Les étiquettes au-dessus des flèches indiquent les trames envoyées et les chiffre entre parenthèses indiquent le contenu des trames en hexadécimal.

Dans les explications qui suivent une trame envoyée en diffusion (broadcast) n'a pas besoin de l'adresse spécifique du récepteur, par contre, dans le cas d'une monodiffusion (unicast) l'adresse du récepteur doit être connue. Le protocole iSN ne définit pas de procédure pour obtenir cette adresse, cette procédure est spécifique à l'implémentation et l'adresse est en principe obtenue en utilisant un protocole d'une autre couche réseau.

Voici comment fonctionne la procédure de connexion en détail :

1. Le client envoie une trame ISN_MSG_SEARCH_SINK en diffusion. Dans le cas sur la figure le IsnClient cherche un sink de type température (01) mais le deuxième octet pourrait être différent. Après avoir envoyé la trame le client attend une réponse pendant la période de temps indiquée par le paramètre ISN_CLIENT_SEARCH_TIMEOUT, s'il ne reçoit pas de réponse alors il va renvoyer la trame le nombre de fois spécifié dans le paramètre ISN_CLIENT_SEARCH_RETRY.
2. Le serveur reçoit la trame ISN_MSG_SEARCH_SINK, il compare le deuxième octet (type d'équipement) avec son propre paramètre type d'équipement. Si les deux sont égaux le serveur répond en monodiffusion avec une trame ISN_MSG_SEARCH_SINK_ACK s'ils ne sont pas égaux le serveur ignore la trame.
3. Le client reçoit la trame ISN_MSG_SEARCH_SINK_ACK et envoie en monodiffusion une trame ISN_MSG_CONNECT. Après avoir envoyé la trame le client attend une réponse pendant la période de temps indiquée par le paramètre ISN_CLIENT_CONNECT_TIMEOUT, s'il ne reçoit pas de réponse alors il va renvoyer la trame le nombre de fois spécifié dans le paramètre ISN_CLIENT_CONNECT_RETRY.
4. Le serveur reçoit la trame ISN_MSG_CONNECT et envoie à son tour une trame ISN_MSG_CONFIG en monodiffusion contenant les paramètres de configuration initiaux du client. Les paramètres qui sont envoyés initialement sont laissés à la discrétion du développeur. Après avoir envoyé la trame le serveur attend une réponse pendant la période de temps indiquée par le paramètre ISN_SERVER_CONFIG_TIMEOUT, s'il ne reçoit pas de réponse alors il va renvoyer la trame le nombre de fois spécifié dans le paramètre ISN_SERVER_CONFIG_RETRY. S'il ne reçoit toujours pas de réponse après cela il abandonne la procédure de connexion.
5. Le client reçoit la trame ISN_MSG_CONFIG, la lit, puis enregistre ses paramètres selon son contenu. Ensuite le client envoie en monodiffusion une trame ISN_MSG_CONFIG_ACK au serveur. À ce moment-là la procédure de connexion est conclue pour le client.
6. Le serveur reçoit la trame ISN_MSG_CONFIG_ACK. À ce moment-là il enregistre l'adresse du client dans sa liste de clients et la procédure de connexion est conclue du côté serveur.

3.4.5 Procédure de keepalive

Afin de réduire le trafic réseau, iSN ne demande pas d'accuser réception des trames ISN_MSG_MEASURE. En conséquence, les clients envoient leurs mesures en utilisant une stratégie « envoyer-oublier ». Pour cette raison le protocole doit définir une procédure pour s'assurer que la connexion client-serveur tient toujours à défaut que les clients envoient leurs mesures sans qu'aucun sink ne les reçoivent et ce sans s'en rendre compte.

Cette procédure s'appelle la procédure de keepalive et elle a pour but de vérifier la bonne « santé » de la connexion client-serveur. Elle est exécutée par le client à tous les intervalles de temps spécifiés par le paramètre ISN_CLIENT_CONFIG_KEEP_ALIVE.

Plus la valeur de ce paramètre est petite, plus les problèmes de connexion seront détectés rapidement et plus le trafic réseau sera grand. En contraste, plus la valeur de ce paramètre est grande moins rapidement seront détectés les problèmes de connexion et moins grand sera le trafic réseau.

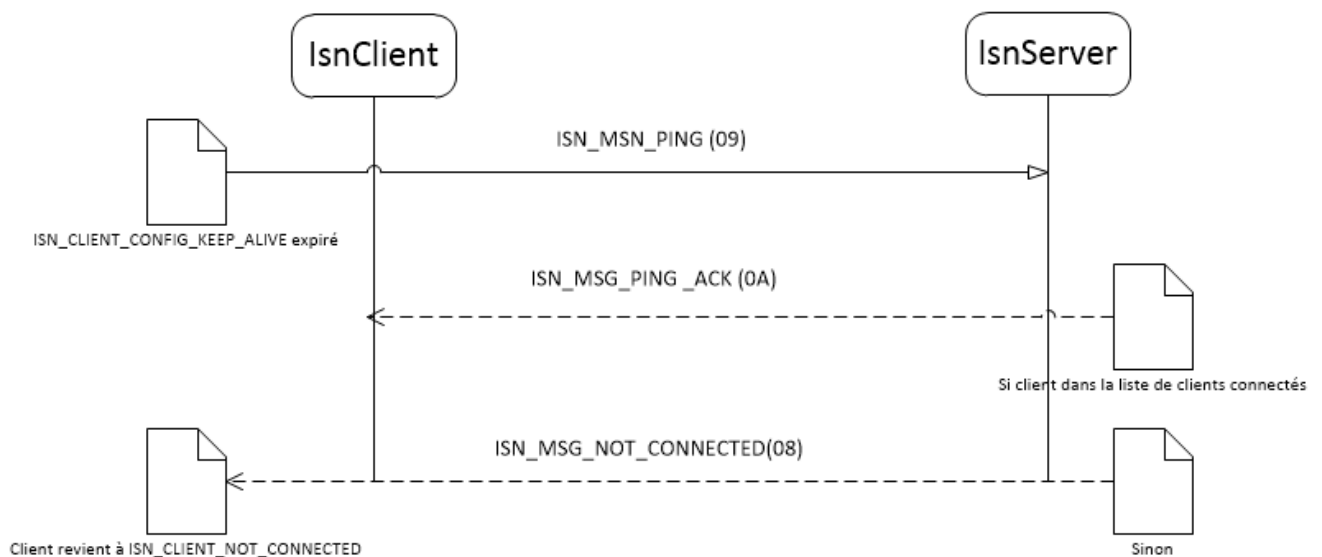


Figure 9 Procédure de keepalive

La procédure de keepalive est illustrée par la Figure 9. Le sens des flèches indique le sens de l'envoi des trames. Les étiquettes au-dessus des flèches indiquent les trames envoyées. Les chiffres entre parenthèse indiquent la valeur hexadécimale des trames. Les flèches pointillées indiquent un choix.

Voici une description du déroulement de la procédure de keepalive :

1. L'intervalle de temps spécifié par le paramètre `ISN_CLIENT_CONFIG_KEEP_ALIVE` est dépassé du côté client. **À noter : le chronomètre comptant cet intervalle doit être remis à zéro si le client reçoit toute trame de son serveur par exemple une trame de configuration. De cette manière on évite de générer trop de trafic réseau inutilement.**
2. Le client envoie une trame `ISN_MSG_PING` en monodiffusion. Après avoir envoyé la trame le client attend une réponse pendant la période de temps indiquée par le paramètre `ISN_CLIENT_PING_TIMEOUT`, s'il ne reçoit pas de réponse alors il va renvoyer la trame le nombre de fois spécifié dans le paramètre `ISN_CLIENT_PING_RETRY`. Si le serveur n'a toujours pas répondu après ça, le client se déconnecte.
3. Le serveur reçoit la trame `ISN_MSG_PING` et lit l'adresse source de la trame. Il vérifie ensuite si l'adresse source correspond à l'adresse d'un de ses clients dans sa liste de clients. Si c'est le cas le serveur envoie une trame `ISN_MSG_PING_ACK` en monodiffusion au client dans le cas contraire, il envoie une trame `ISN_MSG_NOT_CONNECTED` en monodiffusion au client.
4. Si le client reçoit une trame `ISN_MSG_PING_ACK` alors le client poursuit ses opérations sans changement. Autrement, s'il reçoit une trame `ISN_MSG_CONNECTED`, le client se déconnecte de ce sink.

3.4.6 États-transitions

`IsnClient` et `IsnServer` ont été modélisés comme des machines à états finis. C'est-à-dire que les tâches qu'ils accomplissent varient en fonction de l'état dans lequel ils se trouvent. La réception ou l'envoi de trames permettent la transition d'état.

3.4.6.1 IsnClient



Figure 10 États transitions IsnClient

La Figure 10 illustre le diagramme états-transitions d'IsnClient. Les rectangles représentent les états et les flèches les transitions. Les étiquettes au-dessus des flèches indiquent les conditions pour que la transition soit faite. Nous expliquons maintenant en détails chaque état.

ISN_CLIENTSTATE_NOT_CONNECTED

Cet état est l'état initial du client. Comme son nom l'indique, le client se trouve dans cet état lorsqu'il n'est pas connecté à un serveur.

Le seul but de cet état est d'envoyer une trame ISN_MSG_SEARCH_SINK dans l'espoir de trouver un serveur avec lequel initier une connexion.

L'envoi de la trame ISN_MSG_SEARCH_SINK dans cet état provoque la transition vers l'état ISN_CLIENTSTATE_SEARCH_SENT.

ISN_CLIENTSTATE_SEARCH_SENT

Le client entre dans cet état lorsqu'il a envoyé une trame ISN_MSG_SEARCH_SINK.

Le but de cet état est d'attendre une réponse d'un sink. Elle a aussi la responsabilité de renvoyer la trame ISN_MSG_SEARCH_SINK jusqu'à ISN_CLIENT_SEARCH_RETRY fois lorsque le délai ISN_CLIENT_SEARCH_TIMEOUT est atteint.

Le client retourne à l'état ISN_CLIENTSTATE_NOT_CONNECTED s'il dépasse ISN_CLIENT_SEARCH_RETRY fois le délai spécifié par le paramètre ISN_CLIENT_SEARCH_TIMEOUT sans recevoir de trame ISN_MSG_SEARCH_SINK_ACK de la part d'un sink.

Le client effectue une transition vers l'état ISN_CLIENT_SINK_FOUND s'il reçoit une trame ISN_MSG_SEARCH_SINK_ACK de la part d'un sink.

ISN_CLIENTSTATE_SINK_FOUND

Le client entre dans cet état lorsqu'il reçoit une trame ISN_MSG_SEARCH_SINK_ACK de la part d'un sink. Cet état indique que le client a réussi à trouver un sink.

La tâche à effectuer par cet état est simplement d'envoyer une trame ISN_MSG_CONNECT à l'adresse du sink qui a été trouvé à l'état précédent. Lorsqu'elle est envoyée le client effectue une transition à l'état ISN_CLIENTSTATE_CONNECT_SENT.

ISN_CLIENTSTATE_CONNECT_SENT

Le client entre dans cet état lorsqu'il envoie une trame ISN_MSG_CONNECT à un sink. Cet état sert à indiquer qu'une demande de connexion a été envoyée.

Le but de cet état est d'attendre une trame ISN_MSG_CONFIG de la part du sink. Elle a aussi la responsabilité de renvoyer la trame ISN_MSG_CONNECT jusqu'à ISN_CLIENT_CONNECT_RETRY fois lorsque le délai ISN_CLIENT_CONNECT_TIMEOUT est atteint.

Le client retourne à l'état `ISN_CLIENTSTATE_NOT_CONNECTED` s'il dépasse `ISN_CLIENT_CONNECT_RETRY` fois le délai spécifié par le paramètre `ISN_CLIENT_CONNECT_TIMEOUT` sans recevoir de trame `ISN_MSG_CONFIG` de la part d'un sink.

Le client passe à l'état `ISN_CLIENTSTATE_CONFIG_RECEIVED` lorsqu'il reçoit une trame `ISN_MSG_CONFIG` de la part du sink.

ISN_CLIENTSTATE_CONFIG_RECEIVED

Le client entre dans cet état lorsqu'il reçoit une trame `ISN_MSG_CONFIG` de la part du sink. Cet état indique que le client a reçu ses paramètres de configuration initiaux par le serveur.

Le but de cet état est que le client applique les paramètres de configuration reçus. Lorsque cela est fait, la trame `ISN_MSG_CONFIG_ACK` est envoyée au sink.

Lorsque la trame `ISN_MSG_CONFIG_ACK` est envoyée le client passe à l'état `ISN_CLIENTSTATE_CONNECTED`.

ISN_CLIENTSTATE_CONNECTED

Le client entre dans cet état lorsqu'il a envoyé la trame `ISN_MSG_CONFIG_ACK` au sink. Cet état sert à indiquer que le client a maintenant établi une connexion avec un sink.

Plusieurs tâches sont effectuées dans cet état :

1. La procédure de keepalive est effectuée. C'est-à-dire qu'un chronomètre compte jusqu'à `ISN_CLIENT_CONFIG_KEEP_ALIVE`. Lorsque le temps est atteint le client enclenche la procédure de keepalive décrite au point 3.4.5. Trois choses peuvent arriver suite à la procédure de keepalive. Premièrement, le sink peut répondre avec une trame `ISN_PING_ACK`, dans ce cas-là le client reste à l'état `ISN_CLIENTSTATE_CONNECTED` et le chronomètre du keepalive est remis à zéro. Deuxièmement, le serveur peut répondre avec une trame `ISN_MSG_NOT_CONNECTED` dans ce cas-là le client doit revenir à l'état `ISN_CLIENTSTATE_NOT_CONNECTED`. Troisièmement le sink peut ne pas répondre du tout, si tel est le cas le client doit retourner à l'état `ISN_CLIENTSTATE_NOT_CONNECTED`.
2. Le client doit être prêt à recevoir une trame `ISN_MSG_CONFIG` à tout moment. Dans le cas où il reçoit cette trame, il doit changer ses paramètres par ceux fournis dans celle-ci. Finalement il doit envoyer une trame `ISN_MSG_CONFIG_ACK` au sink pour accusé réception de la trame de configuration.

3. IsnClient pour un capteur doit enclencher un chronomètre qui compte jusqu'à ISN_CONFIG_SAMPLING. Lorsque le temps est atteint le capteur doit générer un délai aléatoire entre 0 et ISN_CONFIG_SAMPLING_DELAY. Lorsqu'à son tour le délai aléatoire est atteint le capteur doit obtenir sa mesure courante et l'envoyer à son sink via une trame ISN_MSG_MEASURE. La manière d'obtenir la mesure n'est pas spécifiée par le protocole et est spécifique à l'implémentation.
4. IsnClient pour un actionneur doit être prêt à recevoir des trames ISN_MSG_COMMAND de la part de son sink et à agir en conséquence selon le contenu de ces trames.

3.4.6.2 IsnServer

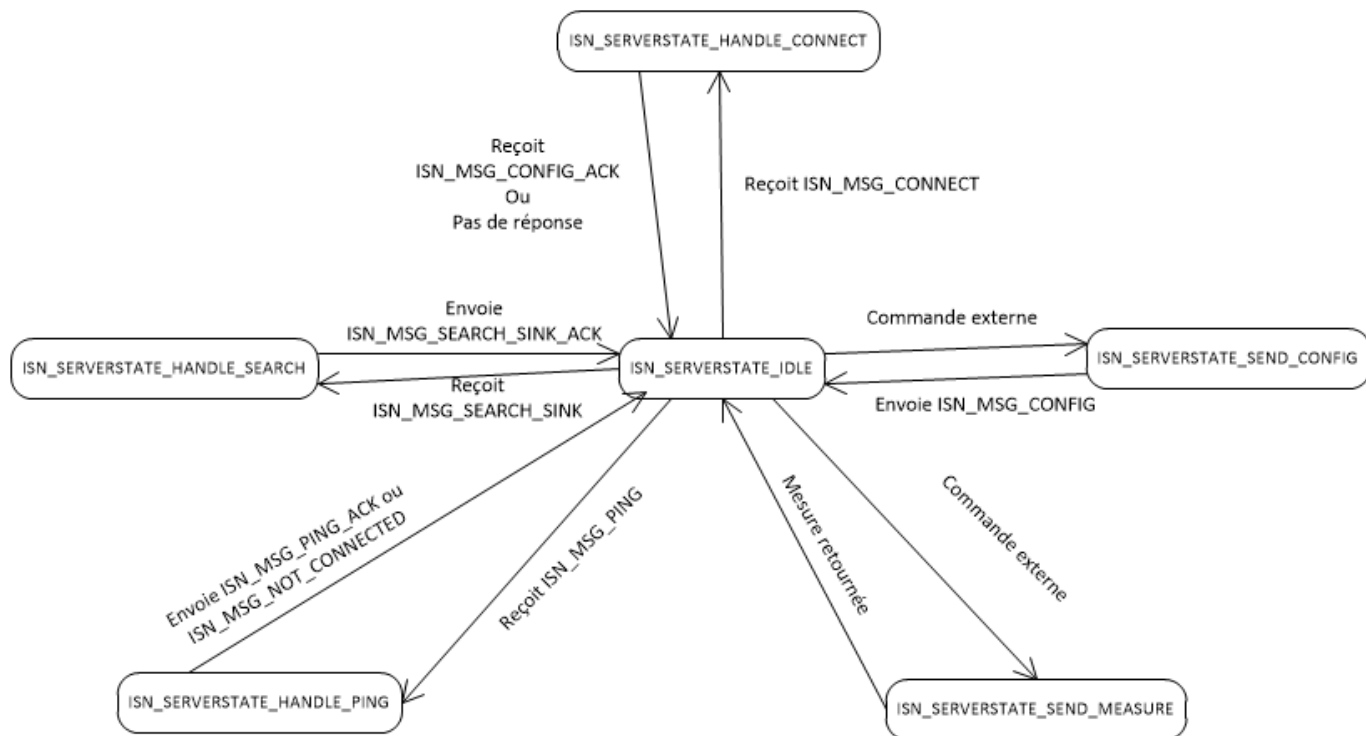


Figure 11 États-Transitions IsnServer

Nous nous concentrons maintenant sur les différents états d'IsnServer. Le diagramme états-transitions correspondant est illustré par la Figure 11.

Le serveur adopte une approche plutôt réactive. C'est-à-dire qu'il est à l'écoute et réagit seulement lorsqu'il est sollicité par un acteur externe.

Nous énumérons maintenant chaque état en détail :

ISN_SERVERSTATE_IDLE

C'est l'état initial. Dans cet état le serveur est passif, il ne fait rien du tout mais reste tout de même à l'écoute des événements. Le serveur écoute pour les événements suivants :

1. La réception d'une trame ISN_MSG_SEARCH_SINK.
2. La réception d'une trame ISN_MSG_CONNECT.
3. La réception d'une trame ISN_MSG_PING.
4. La réception d'une requête pour obtenir la mesure courante.
5. La réception d'une requête pour mettre à jour les paramètres de configuration de ses clients.

ISN_SERVERSTATE_HANDLE_CONNECT

Le serveur entre dans cet état lorsqu'il est dans l'état ISN_SERVERSTATE_IDLE et reçoit une trame ISN_MSG_CONNECT. Le but de cet état est d'envoyer une trame ISN_MSG_CONFIG au client en accusé de réception de la trame de connexion.

Une fois la trame de configuration envoyée le serveur attend un accusé de réception sous la forme d'une trame ISN_MSG_CONFIG_ACK. S'il ne reçoit pas de réponse dans le délai décrit par le paramètre ISN_SERVER_CONFIG_TIMEOUT il renvoie la trame ISN_MSG_CONFIG jusqu'à ISN_SERVER_CONFIG_RETRY fois.

Le serveur retourne à l'état ISN_SERVER_STATE_IDLE qu'il reçoive l'accusé ou non la différence est que dans le cas où il le reçoit il doit ajouter le client à sa liste de client autrement il n'ajoute pas le client à sa liste de clients.

À noter que le serveur doit répondre à une demande de connexion d'un client même si celui est déjà dans sa liste de clients.

ISN_SERVERSTATE_HANDLE_SEARCH

Le serveur entre dans cet état lorsqu'il est dans l'état ISN_SERVER_STATE_IDLE et reçoit une trame ISN_MSG_SEARCH_SINK.

Dans cet état, le serveur compare le paramètre « type d'équipement » de la trame ISN_MSG_SEARCH_SINK à son propre type d'équipement. Si les deux correspondent alors le sink répond avec une trame ISN_MSG_SEARCH_SINK_ACK sinon il ignore simplement la trame reçue.

Dans les deux cas le serveur retourne à l'état ISN_SERVERSTATE_IDLE après avoir effectué ce traitement.

ISN_SERVERSTATE_HANDLE_PING

Le serveur entre dans cet état lorsqu'il est dans l'état ISN_SERVER_STATE_IDLE et reçoit une trame ISN_MSG_PING.

Dans cet état le serveur vérifie si l'adresse source de la trame ping reçue figure dans sa liste de client. Si c'est le cas il envoie une trame ISN_MSG_PING_ACK en réponse, sinon il envoie une trame ISN_MSG_NOT_CONNECTED en réponse.

Dans les deux cas il retourne à l'état ISN_SERVERSTATE_IDLE après avoir fait ce traitement.

ISN_SERVERSTATE_SEND_MEASURE

Le serveur entre dans cet état lorsque du code externe au protocole demande d'obtenir la mesure courante. Cet état est un peu particulier car il est atteint par une action qui vient non pas d'IsnClient mais du code externe au protocole. L'interface pour accéder à la mesure courante n'est pas spécifiée par le protocole mais un appel de fonction publique par exemple : float getMesure() est un moyen efficace d'implémenter cette fonctionnalité.

La manière de calculer la mesure retournée sera décrite dans une section suivante. Une fois que la mesure est retournée à l'appelant le serveur retourne à l'état ISN_SERVERSTATE_IDLE.

ISN_SERVERSTATE_SEND_CONFIG

Le serveur entre dans cet état lorsque du code externe au protocole demande d'envoyer de nouveaux paramètres de configuration aux IsnClients. Comme l'état précédent, cet état est un peu particulier car il est atteint par une action qui vient non pas d'IsnClient mais du code externe au protocole. L'interface pour accéder à cet état n'est pas spécifiée par le protocole mais un appel de fonction publique par exemple : void sendConfig(config) est un moyen efficace d'implémenter cette fonctionnalité.

Cet état a pour responsabilité de prendre les paramètres de configuration en entrée, les encapsuler dans une trame ISN_MSG_CONFIG et de les envoyer à **tous** les clients dans sa liste de clients.

Il procède en envoyant à un client à la fois en attendant une trame ISN_MSG_CONFIG_ACK en réponse. Lorsqu'il reçoit cette trame il envoie la trame ISN_MSG_CONFIG au prochain client dans la liste. S'il ne reçoit rien après avoir attendu pendant un délai ISN_SERVER_CONFIG_TIMEOUT pour ISN_SERVER_CONFIG_RETRY fois alors il retire le client qui ne répond pas de sa liste de client

et passe au prochain. Une fois cette procédure terminée le serveur retourne à l'état ISN_SERVERSTATE_IDLE.

3.4.7 Calcul de la mesure du côté serveur

La gestion des mesures est un thème central du protocole iSN, nous dédions donc cette section à la définition. L'interface par laquelle est obtenue la mesure du côté serveur n'est pas spécifiée par le protocole, en revanche, la manière de calculer cette mesure est spécifiée ici. Nous recommandons le codage d'une fonction publique float getMesure() pour agir en guise d'interface pour obtenir la mesure mais nous laissons la liberté au développeur d'implémenter l'interface qui fonctionne la mieux avec son implémentation.

Nous rappelons que le sink est connecté à plusieurs clients. Il obtiendra donc fort probablement des mesures différentes des différents clients. Pour combiner ces valeurs en une seule, le serveur fait la moyenne de toutes ces mesures.

3.4.7.1 Structure de donnée pour le client

Le serveur devrait garder les informations suivantes pour chaque client dans sa liste de client:

1. Le nombre de fois que le client a échoué d'envoyer sa mesure à temps pour que le sink l'utilise dans le calcul de la moyenne.
2. La dernière mesure reçue de la part de ce client.
3. Une valeur qui indique si la dernière mesure est nouvelle ou pas. Une mesure est considérée nouvelle si elle n'a pas encore été utilisée dans le calcul d'une moyenne.

3.4.7.2 Algorithmes pour le calcul de la moyenne

Plusieurs fonctions sont utilisées pour calculer la moyenne des mesures nous présentons ici le pseudocode des différentes fonctions nécessaires.


```

AllMeasuresArrived()

    PourChaque IsnClientInfo dans listeClients
        Si Non newValue Alors
            Retourner Faux
        Fin Si
    Fin PourChaque
    Retourner Vrai

```

Figure 12 Pseudocode AllMeasuresArrived()

La Figure 12 illustre le pseudocode pour la fonction prédicat AllMeasuresArrived() cette fonction sert à vérifier si la mesure courante de tous les clients dans la liste de clients est à jour. Elle retourne vrai si toutes les mesures courantes des clients sont considérées nouvelles.

```

ApplyPenalties()

    PourChaque IsnClientInfo dans listeClients
        Si Pas newValue Alors
            incrémenter nbMissed
            Si nbMissed >= ISN_SERVER_CONFIG_MISS_LIMIT Alors
                Retirer client de la liste
            Fin Si
        Fin Si
    Fin PourChaque

```

Figure 13 Pseudocode ApplyPenalties()

La Figure 13 illustre le pseudocode pour la fonction ApplyPenalties() cette fonction sert à appliquer une pénalité aux clients qui n'ont pas envoyé une mesure à jour. Elle passe au travers de toute la liste de client en regardant si la mesure du client est une nouvelle mesure, si ce n'est pas le cas elle incrémente le compteur représentant le nombre de fois que le client a échoué d'envoyer une nouvelle valeur. Si ce compteur est plus grand ou égal à la valeur spécifiée par le paramètre ISN_SERVER_CONFIG_MISS_LIMIT alors le client est retiré de la liste de client du serveur.

Important : Le compteur représentant le nombre de fois que le client a échoué d'envoyer une nouvelle valeur à temps doit être remise à zéro lorsque le serveur reçoit une trame ISN_MSG_MEASURE de ce client. On compte ici le nombre d'échecs **successifs** et non pas le nombre d'échecs total.

```
ComputeAverage()

Total = 0
Somme = 0

PourChaque IsnClientInfo dans listeClients
    Si newValue Alors
        Incrémenter Total
        Somme = somme + mesure
        newValue = Faux
    Fin Si
Fin PourChaque

Si Total = 0 Alors
    Retourner Erreur
Sinon
    Retourner Somme/Total
Fin Si
```

Figure 14 Pseudocode ComputeAverage()

La Figure 14 représente le pseudocode pour la fonction ComputeAverage(). Dans cette figure la variable Total représente le nombre total de mesures prises en considération lors du calcul de la moyenne. La variable somme représente un accumulateur pour les valeurs des mesures.

On boucle au travers de tous les clients dans la liste de client si leur mesure courante est à jour alors on la prend en considération dans le calcul de la moyenne sinon on ne touche pas à cette valeur car elle pourrait fausser la valeur de la moyenne calculée.

Il ne faut pas oublier que toutes les valeurs qui sont utilisés dans le calcul de la moyenne sont maintenant considérées comme des valeurs désuètes.

À la fin de la boucle si le nombre de mesures ayant participé au calcul est toujours zéro alors toutes les mesures des clients étaient de vieilles valeurs il faut donc renvoyer une valeur d'erreur.

Cette valeur n'est pas définie par le protocole mais nous recommandons la valeur d'une mesure qui ne fait pas de sens dans le contexte.

```
GetMeasure()

Si Pas AllMeasuresArrived() Alors
    Attendre pour une période de grâce (ISN_SERVER_CONFIG_GRACE_DELAY)
    Si Pas AllMeasuresArrived() Alors
        applyPenalties()
    Fin Si
Fin Si

Mesure = ComputeAverage()
Retourner Mesure
```

Figure 15 Pseudocode GetMeasure()

La Figure 15 montre le pseudocode pour la fonction GetMeasure(). Cette fonction est la fonction publique qui est exposée au code externe. Toutes les autres fonctions présentées étaient des fonctions privées utilisées par GetMeasure().

Lorsque GetMeasure() est appelée elle vérifie d'abord si toutes les mesures des clients sont à jour à l'aide de la fonction AllMeasuresArrived(). Si elle sont toutes arrivées alors on calcul la moyenne des mesures via la fonction ComputeAverage() et on retourne sa valeur à l'appelant.

Si certaines mesures ne sont pas arrivées alors on donne une chance aux clients en attendant une période de temps spécifiée par le paramètre ISN_SERVER_CONFIG_GRACE_DELAY (voir section 3.4.3.1).

Important : Le serveur doit continuer de lire des trames pendant le délai de grâce sinon il ne recevra pas les nouvelles mesures qui pourraient arriver.

Si après ce délai certaines mesures ne sont toujours pas arrivées alors on applique une pénalité aux clients fautifs à l'aide de la fonction applyPenalties().

3.5 Notre implémentation d'iSN

Avec la définition du protocole derrière nous, nous expliquons maintenant comment nous avons implémenté le protocole :

Premièrement, nous avons créé deux projets; Le premier projet s'appelle MQTT-SN-Client et contient l'implémentation d'IsnServer. Son nom vient du fait que le projet contient aussi une implémentation d'un client MQTT-SN, néanmoins, l'implémentation d'iSN a été faite de manière à séparer le plus possible ces deux protocoles pour garder une bonne modularité. Ce projet contient le code qui sera exécuté par la carte microcontrôleur freescale du composant sink.

Le deuxième projet s'appelle IsnClientTest. Comme son nom l'indique ce projet contient l'implémentation d'iSN côté client. C'est un projet pour tester l'implémentation d'IsnClient c'est-à-dire qu'il contient du code qui implémente un faux capteur qui donne une mesure constante. On peut aussi s'en servir pour comprendre comment intégré IsnClient avec le reste du code du capteur. Ce code est celui qui s'exécutera sur la carte microcontrôleur freescale du composant capteur.

Le code est un mélange du langage C et du langage C++ et est compilé en code assembleur ARM pour processeur Cortex M0. Nous utilisons l'IDE Kinetis Design Studio de NXP comme plateforme pour faire la programmation embarquée. L'outil de développement Processor Expert permet d'automatiquement générer le code de bas niveau pour accéder aux composantes de la carte nous permettant de nous concentrer sur le code important.

3.5.1 Structure des projets

Ici nous allons décrire le rôle des répertoires et des fichiers importants pour les deux projets en commençant par MQTT-SN-Client.

3.5.1.1 MQTT-SN-Client

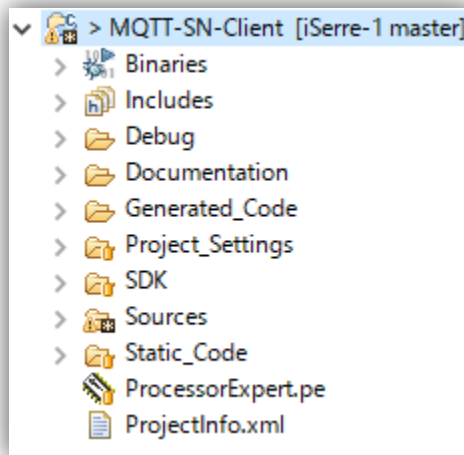


Figure 16 Structure du projet MQTT-SN-Client

La Figure 16 montre la structure du projet MQTT-SN-Client. Deux répertoires sont à retenir :

1. **Generated_Code** : Contient tout le code qui a été généré par Processor Expert. Il est jamais nécessaire de modifier ce code toutefois il est parfois nécessaire d'inclure les fichiers d'en-tête (.h) si le code utilise ces composantes.
2. **Sources** : Contient tout notre code. Tout nouveau code devrait être ajouté à ce répertoire.

Répertoire Sources

Nous présentons maintenant en plus de détails le répertoire sources et ses fichiers.

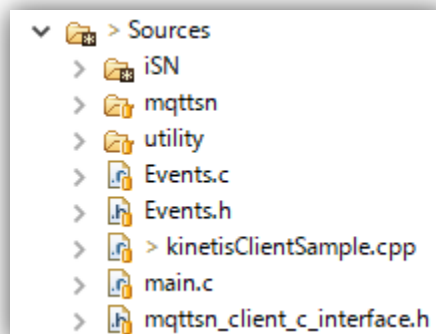


Figure 17 Le répertoire Sources

La Figure 17 illustre la structure du répertoire sources. Nous décrivons ci-dessous les sous-répertoires et fichiers du répertoire sources :

1. **Isn** : Ce répertoire contient les fichiers sources en lien avec l'implémentation d'IsnServer.
2. **Mqttsn** : Ce répertoire contient les fichiers sources en lien avec l'implémentation de MQTT-SN-Client.
3. **Event.c** : Ce fichier est un fichier contenant les gestionnaires d'évènements déclenchés par les interruptions du processeur. Il est important pour nous pour l'interruption déclenché par l'horloge interne. **Le code suivant doit être présent dans ce fichier sinon aucun chronomètre ne fonctionnera dans le code.**

```
void PIT_IRQHandler(void)
{
    /* Clear interrupt flag.*/
    PIT_HAL_ClearIntFlag(g_pitBase[PIT_Timer0_IDX], PIT_Timer0_CHANNEL);
    /* Write your code here ... */
    Clock_incrementMsCounter();
}
```

4. **KinetisClientSample.cpp** : Ce fichier contient du code pour initialiser MQTT-SN. Il contient aussi le code pour les communications de MQTT-SN vers IsnServer. Par exemple, pour envoyer des trames de configuration sur demande. L'état du serveur ISN_SERVERSTATE_SEND_CONFIG décrite à la section 3.4.6.2 est d'ailleurs déclenché par le code dans ce fichier.
5. **Main.c** : C'est le point d'entrer du code généré par processeur expert. Contrairement au reste du code processeur expert on peut ajouter nos propres include dans ce fichier et nos propres appels de fonction mais ils doivent être faits dans les sections mises en commentaire par Processeur Expert.
6. **Mqttsn_client_c_interface.h** : Ce fichier d'en-tête déclare une fonction qui a comme attribut « extern C » cela permet à cette fonction C++ d'être appelé à partir de code C. Elle sert de point de transition entre C et C++ et elle est nécessaire car Processeur Expert ne génère que du code C.

Répertoire iSN

Voici le répertoire iSN en plus de détails :

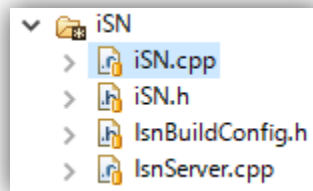


Figure 18 Répertoire iSN

iSN.cpp : Contient le code pour les objets partagés entre IsnServer et IsnClient par exemple les trames et des fonctions de débogage.

IsnBuildConfig.h : Fichier d'en-tête qui contient les paramètres de compilation pour IsnServer. Commenté ou dé-commenter les instructions define permet de changer les paramètres. À noter que l'instruction ISN_DEBUG active toutes les sorties à la console série pour faciliter le débogage.

IsnServer.cpp : Contient le code implémentant IsnServer. À noter que les définitions pour iSN.cpp et IsnServer.cpp se trouvent toute deux dans le fichier iSN.h.

Répertoire mqttsn

Bien qu'il soit hors de la portée du protocole, le répertoire mqttsn contient tout de même certains fichiers utilisés par iSN. Nous allons donc le présenter en plus de détails :

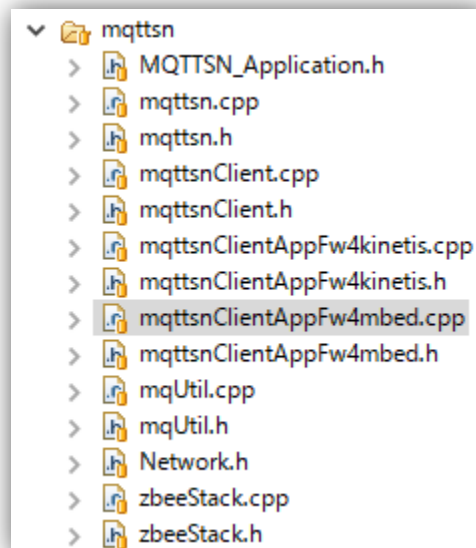


Figure 19 Répertoire mqttsn

MQTTSN_Application.h : Cet en-tête définit les structures pour les paramètres de configuration des Xbees plus spécifiquement XbeeConfig.

mqttsnClientAppFw4kinetis.cpp : Ce fichier contient la boucle principale pour le code du sink. Cette boucle se trouve dans la fonction mqttsnClientAppMain() et fait simplement exécuter iSNServer suivi de mqtt-sn-client une fois par itération.

mqUtil.h : Ce fichier d'en-tête contient la définition de la classe XTimer qui est emprunté dans notre implémentation d'iSN pour implémenter les chronomètres.

zbeeStack.h : Ce fichier d'en-tête contient la définition de toutes les classes nécessaires pour faire la manipulation de trames api de DigiMesh ainsi que l'envoi et la réception de données sur un UART. Nous la réutilisons dans notre implémentation d'IsnServer. Nous avons ajouté la fonction getIsnType() à la classe NWResponse car la fonction getType() fonctionne pour une trame mqtt mais pas pour une trame iSN.

Répertoire utility

Le dernier répertoire s'appelle utility et contient des fonctions utilitaires utilisées un peu partout dans le code. Voici sa structure :

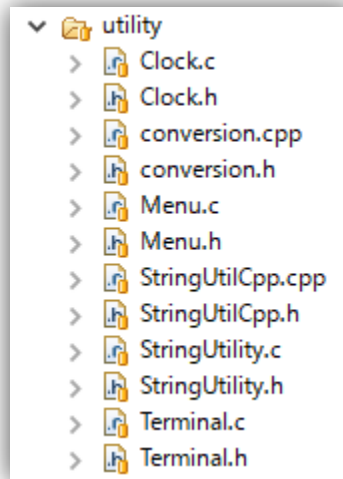


Figure 20 Répertoire utility

Clock.c : Contient des fonctions utilitaires et une variable globale compteur pour l'horloge. Le fonctionnement de la classe XTimer qui implémente les chronomètres est dépendant des fonctions dans ce fichier.

Important : Le compteur devrait être initialiser/réinitialiser à la valeur de 1 et non 0.

Conversion.cpp : Contient des fonctions de conversion entre types de données.

StringUtilCpp.cpp : Contient des fonctions utilitaires C++ en lien avec les chaînes de caractères (string).

StringUtility.c : Contient des fonctions utilitaires C en lien avec les chaînes de caractères (string).

3.5.1.2 IsnClientTest

Nous nous concentrons maintenant sur la structure du projet IsnClientTest. La plupart des fichiers sont les mêmes alors nous allons seulement présenter les nouveaux fichiers, vous pouvez vous référer à la section précédente pour connaître l'utilité des autres fichiers :

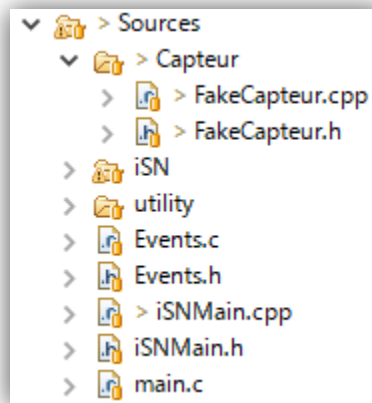


Figure 21 Structure du projet IsnClientTest

Capteur : Ce répertoire contient le fichier `FakeCapteur.cpp` dans ce fichier on implémente un capteur virtuel utilisé à des fins de test du protocole. Ce capteur retourne toujours la même mesure.

iSN : Ce répertoire contient l'implémentation d'`IsnClient`. Les seules différences avec le dernier projet sont : Les dépendances qui provenaient du répertoire `mqttSn` ont été bougées dans ce répertoire. Il y a un fichier `IsnClient.cpp` qui remplace le fichier `IsnServer.cpp`.

iSNMain.cpp : Contient la boucle principale du code. Offre un exemple excellent de la manière d'intégrer le code d'`IsnClient` avec le code d'un capteur.

3.5.2 Classes et fonctions importantes

Dans cette section nous passons en revue toutes les classes et fonctions importantes de l'implémentation. Pour garder cette section simple et compacte nous omettons les accesseurs/mutateurs ainsi que les fonctions et variable qui ne sont pas importantes à la compréhension. Nous allons aussi présenter qu'une seule fois les fonctions ou variables qui se répètent. Nous commençons en présentant La classe IsnMessage :

IsnMessage

La classe IsnMessage est la classe parent de tous les type de messages Isn et représente l'implémentation des trames définies à la section 3.4.2. Nous ne présentons pas toutes les classes car il y en a une par trame et leur implémentation est relativement simple à comprendre.

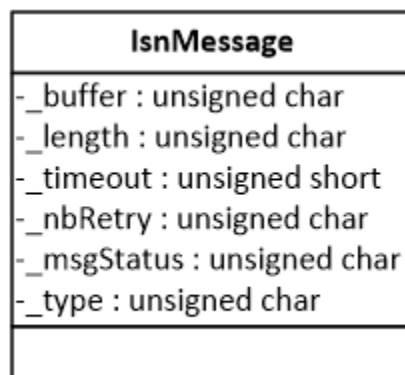


Figure 22 Classe IsnMessage

_buffer : C'est une variable qui est un pointeur vers un tampon contenant les octets de la trame tels que définis à la section 3.4.2.

_length : C'est une variable qui contient la longueur du tampon en octets.

_timeout : Si ce type de trame doit attendre un accusé de réception cette valeur indique pendant combien de temps on attend pour l'accusé en secondes. Elle est utilisée pour implémenter les timeout de la section 3.4.3.1.

_nbRetry : Si ce type de trame doit être renvoyé en l'absence de réception d'un accusé cette variable indique jusqu'à combien de fois renvoyer cette trame. Elle implémente le nombre d'essais décrit à la section 3.4.3.1.

_msgStatus : Un message peut être dans trois états différents :

État	Description
ISN_MSG_STATUS_SEND_REQ	Le message est dans la file d'envoi et demande d'être envoyé au récepteur.
ISN_MSG_STATUS_WAITING	Le message a été envoyé mais est en attente de son accusé de réception.
ISN_MSG_STATUS_COMPLETE	L'accusé de réception pour ce message a été reçu et le cycle de vie du message est maintenant terminé. Il sera retiré de la file d'envoi.

_type : Le type de trame tel que défini à la section 3.4.2

IsnConfiguration

Cette classe encapsule les paramètres de configurations qui peuvent être envoyés à IsnClient via une trame ISN_MSG_CONFIG. Elle contient aussi les paramètres de configuration initiaux lorsque le IsnServer est démarré. Cependant cette classe ne représente pas une trame de configuration. La classe IsnMsgConfig joue ce rôle.

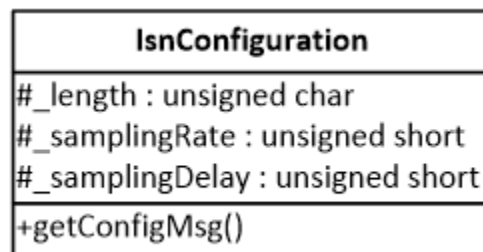


Figure 23 Classe IsnConfiguration

_length : Variable qui contient le nombre de paramètre de configuration. Attention : On parle ici du nombre de paramètres de configuration et non pas du nombre d'octets.

_samplingRate : La valeur du paramètre ISN_CONFIG_SAMPLING décrit à la section 3.4.3.2.

_samplingDelay : La valeur du paramètre ISN_CONFIG_SAMPLING_DELAY décrit à la section 3.4.3.2.

getConfigMsg() : Cette fonction permet de construire une trame ISN_MSG_CONFIG à partir des informations de cette classe.

IsnConfigParam

Cette classe représente un couple nom/valeur dans une trame ISN_MSG_CONFIG tel que défini à la section 3.4.3.2.

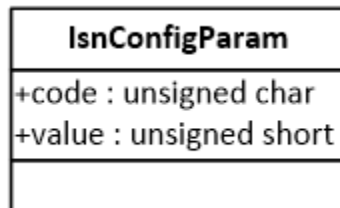


Figure 24 Classe IsnConfigParam

Code : Cette variable représente le nom du paramètre.

Value : Cette variable représente la valeur du paramètre.

NWAddress64

Cette classe représente l'adresse 64 bits d'un module xbee dans le mode API du protocole DigiMesh. Elle est utilisée pour représenter les adresses des IsnClients et IsnServeurs.

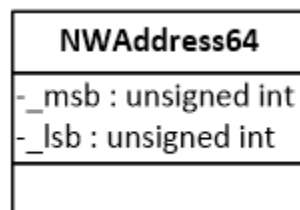


Figure 25 Classe NWAddress64

_msb : Variable qui représente les 32 bits les plus significatifs de l'adresse réseau.

_lsb : Variable qui représente les 32 bits les moins significatifs de l'adresse réseau.

IsnClientInfo

Cette classe est une structure de données qui encapsule toutes les informations qu'IsnServer a besoin de connaître à propos d'un client. Elle est conforme à ce qui a été expliqué à la section 3.4.7.1.

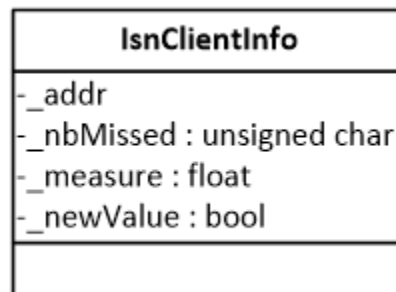


Figure 26 Classe IsnClientInfo

_addr : L'adresse réseau 64 bits du module xbee du client. De type NWAddress64.

_nbMissed : Un compteur qui indique le nombre de fois d'affilé que le client a échoué à envoyer une mesure.

_measure : La dernière mesure reçue de la part de ce client.

_newValue : Un booléen qui indique si la mesure a déjà été utilisée dans le calcul d'une moyenne ou non.

CommonSensor

Une classe abstraite qui représente le capteur. Tout capteur qui interagit avec IsnClient doit hériter de cette classe abstraite et implémenter ses fonctions virtuelles.

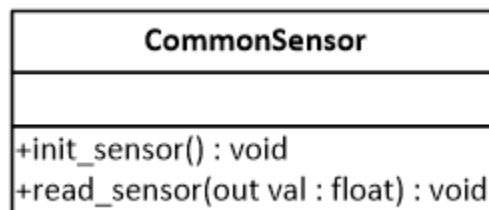


Figure 27 Classe abstraite Common Sensor

Init_sensor() : Lorsque implémenter dans une sous-classe, cette fonction doit contenir le code pour initialiser le capteur. Si l'initialisation n'est pas requise le corps de cette fonction doit être vide.

Read_sensor(float* val) : Lorsque implémenter dans une sous-classe, cette fonction retourne la mesure du capteur dans la variable val dont l'adresse est passée en paramètre.

XTimer

Cette classe est utilisée comme chronomètre pour mesurer les différents délais définis par le protocole :

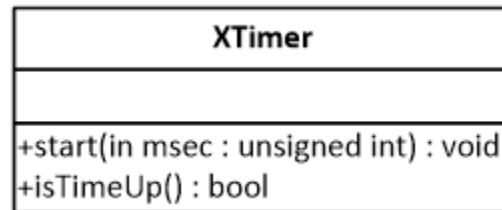


Figure 28 Classe XTimer

Start(uint32_t msec) : Cette fonction démarre le chronomètre, le paramètre msec indique jusqu'à combien de millisecondes compte le chronomètre.

isTimeUp() : Est une fonction prédicat qui retourne vrai si le chronomètre a compté jusqu'au paramètre msec passé à la fonction start et faux autrement.

NWResponse

Cette classe est une structure de données qui représente le format de la trame API de DigiMesh. Elle est passée en paramètre à la fonction de réception de trames. C'est avec elle qu'on peut obtenir l'adresse de l'envoyeur :

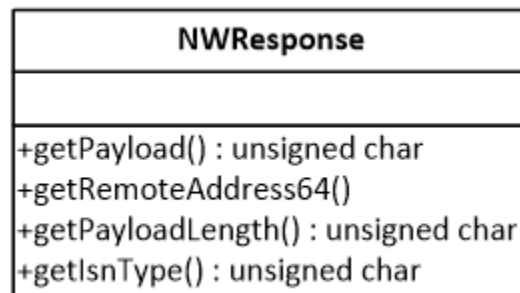


Figure 29 Classe NWResponse

getPayload() : Cette fonction retourne un pointeur vers les données encapsulées par la trame API. Dans notre cas elle retourne un pointeur sur la trame isn.

getPayloadLength() : Retourne la longueur des données encapsulées par la trame API. Dans notre cas retourne la longueur de la trame iSN en octets.

getRemoteAddress64() : Retourne l'adresse xbee 64 bits de l'émetteur de cette trame.

getIsnType() : Retourne le type de trame iSN si les données encapsulée sont une trame iSN.

Network

La classe network implémente le processus d'envoi et de réception de trames API DigiMesh. C'est aussi cette classe qui gère les envoies et réceptions sur les UARTs.

Network
+Network(in instance : unsigned int, in uart_user_config, in uart_state) +send(in xmitData : unsigned char, in dataLength : unsigned char, in SendRequestType) : void +readPacket() : int +setGwAddress(in NWAddress64) : void +setRxHandler(in RxCallback) : void +initialize(in XbeeConfig) : int

Figure 30 Classe Network

Network(instance, uart_user_config, uart_state) : Ce constructeur prend en paramètre le numéro d'instance d'un UART (instance), un pointeur vers la structure de configuration du UART (uart_user_config) et un pointeur vers la structure d'état du UART (uart_state). La classe Network utilisera ce UART pour les communications réseaux. Plus de détails sur la manière d'obtenir ces valeurs seront fournis à la prochaine section.

Send(xmitData, dataLength, sendRequestType) : Cette fonction envoie les octets dans un tampon de données sur le UART (dans notre cas le UART est connecté à un xbee). xmitData est un pointeur vers un tampon contenant la trame à envoyer. dataLength est la longueur de ce tampon en octet. sendRequestType est BcastReq pour envoyer en diffusion (broadcast) et UcastReq pour envoyer en monodiffusion (unicast).

readPacket() : Cette fonction lit une trame dans le tampon de réception du UART. Si une trame est présente dans le tampon alors la fonction de callback réseau est appelée pour pouvoir gérer cette trame, sinon la fonction retourne sans rien faire.

setGwAddress(NWAddress64) : Cette fonction permet de changer l'adresse du module xbee auquel on envoie des trames en mode monodiffusion. Elle prend en paramètre l'adresse du xbee de type NWAddress64.

SetRxHandler(RxCallback) : Cette fonction permet de changer le callback réseau de réception. Ce callback réseau de réception est une fonction qui est appelée lorsqu'une trame est reçue. Il est déclenché par l'appel à la fonction readPacket().

Initialize(XbeeConfig) : C'est une fonction d'initialisation qui doit être appelée avant de commencer à utiliser la classe réseau. Elle prend en paramètre une structure de type XBeeConfig qui elle-même est composée de trois variables, la première est le baudrate du xbee, normalement 5760, le deuxième est le numéro du port série, ce paramètre peut être laissé à 0, le troisième est un paramètre pour le type de périphérique et peut aussi être laissé à 0.

IsnServer

Cette classe implémente le côté serveur du protocole isn et est présente dans le code du sink.

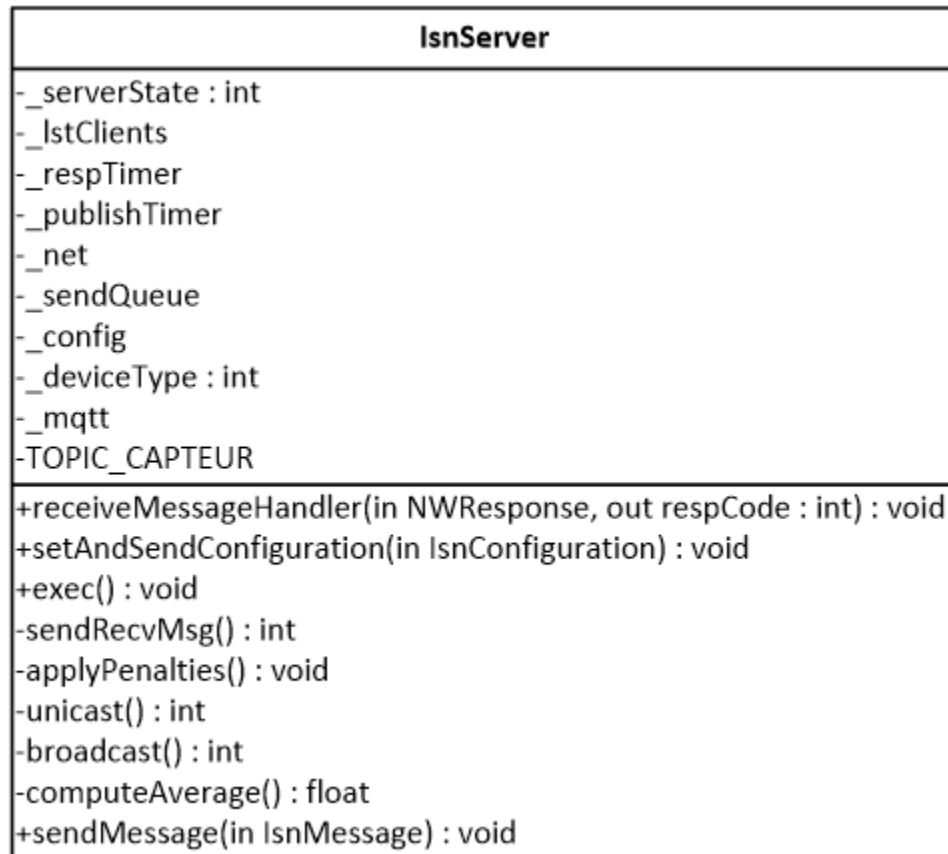


Figure 31 Classe IsnServer

_serverState : Cette variable représente l'état courant du serveur. Les états du serveur sont définis à la section 3.4.6.2.

_lstClients : C'est la liste de client présentement connectés au serveur. Chaque client dans la liste est représenté par un objet de type `IsnClientInfo` et la liste elle-même est de type `std::vector`.

_respTimer : Cette variable est un chronomètre de type `XTimer` qui sert à compter le temps qu'une trame accusé de réception met avant d'être reçue. Il est utilisé pour implémenter les timeout à la section 3.4.3.1.

_publishTimer : Cette variable est un chronomètre de type `XTimer` qui sert à détecter le temps où il faut publier la moyenne des mesures sur le TOPIC_CAPTEUR de mqtt.

_net : Un objet de type `Network` qui est utilisé pour envoyer et recevoir les trames iSN.

_sendQueue : Un objet de type `Queue`, c'est la file d'envoi des trames quand une trame est envoyée on la place d'abord à l'arrière de cette file. La classe `Queue` est une implémentation simple d'une file faite avec une liste interne de type `std::vector`.

_config : Un objet de type `IsnConfiguration`, contient les paramètres de configuration qui seront envoyés à un client lors d'une connexion.

_deviceType : Le type d'équipement qu'est le `IsnServer`. Utilisé lorsqu'une trame `ISN_MSG_SEARCH_SINK` pour vérifier si cette trame s'adresse à ce type de sink. Les types d'équipement sont définis à la section 3.4.3.1

_mqtt : Un pointeur vers un client `MQTT_SN`, est utile pour pouvoir publier périodiquement la moyenne des mesures des capteurs sur un topic MQTT.

TOPIC_CAPTEUR : Une chaîne de caractère qui indique dans quel topic publier la moyenne des mesures.

receiveMessageHandler(NWResponse, respCode) : Le callback réseau pour la réception de trame. C'est là que sont gérées les trames qui sont reçues par `IsnServer`. Le premier paramètre est la trame reçue dans son format API DigiMesh de type `NWResponse`. Le deuxième paramètre est l'adresse d'une variable de type `int` qu'on utilise en paramètre de sortie pour indiquer le code de retour.

setAndSendConfiguration(IsnConfiguration) : Cette fonction permet de changer la variable `_config` du serveur et de systématiquement envoyé la nouvelle configuration à tous les clients dans la liste de clients. C'est le mécanisme de transition pour entrer dans l'état `ISN_SERVERSTATE_SEND_CONFIG` qui est défini à la section 3.4.6.2.

exec () : Cette fonction doit être appelée à chaque itération de la boucle principale pour faire fonctionner le serveur.

SendRecvMsg() : C'est un peu la version privée de `exec()`. Dans cette fonction le protocole vérifie dans quel état il est et selon cet état détermine les actions à entreprendre. Cette fonction fait aussi toujours un appel à `readPacket` pour voir si une trame est disponible dans le tampon du UART.

applyPenalties() : Cette fonction implémente l'algorithme du même nom défini à la section 3.4.7.2

unicast() : Cette fonction permet d'envoyer le `IsnMessage` à l'avant de la file d'envoi en mode monodiffusion. Si cette trame spécifie un timeout (`_timeout`) et un nombre d'essais (`_nbRetry`) alors unicast ne retourne pas tant que l'accusé de réception n'est pas reçu ou que le nombre d'essai n'a pas été épuisé. Si l'accusé est reçu cette fonction retourne un code d'erreur `ISN_RC_NO_ERROR` pour indiquer qu'aucune erreur n'est survenue. Si le nombre d'essais est épuisé alors elle retourne `ISN_RC_RETRY_OVER` comme code d'erreur. Cette fonction retourne aussi `ISN_RC_RETRY_OVER` quand on envoie une trame qui n'attend pas d'accusé il est donc important de regarder le code de retour seulement lorsqu'on envoie une trame qui requiert un accusé de réception.

Broadcast() : Fonctionne exactement de la même manière qu'`unicast()` mais envoie la trame en mode diffusion au lieu du mode monodiffusion.

sendMessage(IsnMessage) : Place une trame iSN à l'arrière de la file d'envoi. Il est important de noter qu'aucun message n'est réellement envoyé par cette fonction. Il faut appeler `unicast()` ou `broadcast()` pour que l'envoi se fasse. Elle prend en paramètre un objet de type `IsnMessage` qui représente une trame iSN.

computeAverage() : Cette fonction calcul la moyenne des mesures des clients et la retourne sous forme de float. Retourne `ISN_RC_INVALID_MEASURE` s'il y a eu erreur dans le calcul de la moyenne. Elle implémente l'algorithme du même nom défini à la section 3.4.7.2.

IsnClient

Cette classe implémente la partie client du protocole iSN elle roule normalement sur un capteur. La partie actionneur n'a pas été implémentée. La plupart des variables et fonctions sont identiques à ceux dans IsnServer. Pour cette raison, nous allons seulement présenter les différences.

IsnClient
<div><div><div>-_clientStatus : int</div><div>-_deviceType : int</div><div>-_net</div><div>-_sendQueue</div><div>-_respTimer</div><div>-_samplingTimer</div><div>-_keepAliveTimer</div><div>-_sensor</div><div>-_config</div></div></div>
<div><div><div>-sendMessage(in IsnMessage) : void</div><div>-broadcast() : void</div><div>-unicast() : void</div><div>-sendRecvMsg()</div><div>-delayTime(in maxTime : unsigned short) : void</div><div>+exec() : void</div><div>+receiveMessageHandler(in NWResponse, out respCode : int) : void</div><div>+sendMeasure(in measure : float) : void</div><div>+setSensor(in CommonSensor) : void</div></div></div>

Figure 32 Classe IsnClient

_clientStatus : Représente l'état courant du client tel que défini à la section 3.4.6.1.

_samplingTimer : Un chronomètre de type XTimer qui sert à savoir quand vient le temps d'échantillonner la valeur du capteur. Fonctionne avec la valeur ISN_CONFIG_SAMPLING définie à la section 3.4.3.1.

_keepAliveTimer : Un chronomètre de type XTimer qui sert à savoir quand vient le temps d'enclencher la procédure de keepalive définie à la section 3.4.5.

_sensor : Un pointeur vers un objet qui hérite de la classe CommonSensor cette variable est utilisée pour échantillonner la mesure du capteur.

delayTime(maxTime) : Permet de générer un délai aléatoire entre 0 et maxTime en secondes. Utilisé pour implémenter le délai indiqué par le paramètre ISN_CONFIG_SAMPLING_DELAY défini à la section 3.4.3.1.

sendMeasure(measure) : Permet d'envoyer la mesure passée en paramètre au IsnServer. La mesure passée en paramètre est de type float.

setSensor(CommonSensor) : Permet de changer la référence au capteur gardée par le client. Cette fonction doit être appelée au moins une fois pour initialiser IsnClient avec un capteur autrement ce pointeur sera NULL et le client ne fonctionnera pas.

3.5.3 Interopérabilité

Pour que notre implémentation d'ISN fonctionne dans son environnement il faut qu'elle communique d'une manière ou d'une autre avec le code du capteur du côté client et avec MQTT-SN du côté serveur. Dans cette section nous expliquons comment nous avons géré l'interopérabilité entre ces différents composants.

3.5.3.1 MQTT-SN

Les fonctions callback du fichier kinetisClientSample.cpp permettent à MQTT-SN d'envoyer des messages à IsnServer. Pour l'instant le seul message reçu de la part de MQTT-SN est une publication dans un topic de configuration pour changer les paramètres dynamiques définis à la section 3.4.3.2. Ce callback fait appel à la fonction setAndSendConfiguration() d'IsnServer pour envoyer les paramètres à IsnServer et ultimement à tous ses IsnClients.

La variable membre _mqtt dans IsnServer est un pointeur vers un client MQTT-SN. Il permet à IsnServer d'envoyer des données à MQTT-SN. Pour l'instant elle est seulement utilisée dans le corps de la fonction sendRecvMsg() pour périodiquement publier la moyenne des mesures des capteurs dans un topic spécifié par la variable membre TOPIC_CAPTEUR.

3.5.3.2 Code du capteur

Nous décrivons ici en détails le minimum de code qui doit être écrit pour intégrer un capteur avec IsnClient. Nous nous serviront du fichier isnMain.cpp du projet IsnClientTest à titre d'exemple.

Avant de commencer, le capteur devrait être codé sous forme de classe C++, hérité de la classe abstraite CommonSensor et implémenter les fonctions init_sensor() et read_sensor().

Similairement au code suivant :

```
class CommonSensor
{
public:
    virtual void init_sensor() = 0;
    virtual void read_sensor(float* val) = 0;
    virtual ~CommonSensor();
};

class FakeCapteur : public CommonSensor
{
public:
    FakeCapteur();
    ~FakeCapteur();
    void exec();
    void init_sensor();
    void read_sensor(float* val);
private:
    float _mesure;
};
```

Figure 33 Sous-classe de CommonSensor

Une fois cela accompli, le code de la fonction principale devrait ressembler à ceci :

```
#define KINETIS
#include "iSNMain.h"
#include "iSN\iSN.h"
#include "uart_xb.h"
#include "iSN\ZbeeStack.h"
#include "Capteur\FakeCapteur.h"

using namespace tomyClient;

void iSNMain()
{
    XBeeConfig theAppConfig = {
        57600,          //Baudrate (bps)
        0,              //Serial PortNo
        0,              //Device (for linux App)
    };

    Network net(UART2_IDX, &uart_xb_InitConfig0, &uart_xb_State);
    net.initialize(theAppConfig);

    IsnClient client(&net, ISN_SENSOR_TEMP);
    FakeCapteur capteur;
    client.setSensor(&capteur);

    while (true)
    {
        client.exec();
        capteur.exec();
    }
}
```

Figure 34 Code minimal pour capteur

1. Vous devez copier tout le répertoire iSN du projet IsnClientTest dans votre projet.
2. Dans le fichier IsnClient .cpp vous allez devoir changer l'include « Capteur/FakeCapteur.h » pour le fichier de définition de votre capteur comme présenté à la Figure 33.
3. Si ce n'est pas déjà fait vous devez ajouter une composante UART avec processeur expert. Dans le cas présent on a nommé ce UART uart_xb. C'est le UART qu'IsnClient utilisera par envoyer et recevoir des trames.
4. Vous devez inclure le fichier d'en-tête du UART ici c'est uart_xb.h.

5. Vous devez inclure le fichier « iSN\ZbeeStack.h » pour accéder aux définitions de la classe Network et la structure XbeeConfig.
6. Vous devez inclure la définition de votre classe capteur similaire à la Figure 33.
7. On déclare une variable de type XbeeConfig pour pouvoir initialiser le réseau. La première valeur est le baudrate du Xbee qui est normalement 57600 mais vous pouvez changer cette valeur dans la mémoire flash du xbee en utilisant XCTU. Les deux autres paramètres peuvent être laissés à zéro.
8. On déclare et initialise une variable de type Network. Le premier paramètre du constructeur est le numéro d'instance du UART. UART1_IDX pour le UART 1 et UART2_IDX pour le UART 2. Le deuxième est un pointeur vers la structure de configuration du UART, le troisième est un pointeur sur la structure d'état du UART.

Figure 35 Paramètres du UART

Comme vous pouvez voir sur la Figure 35 le nom de ces structures est visible en cliquant sur la composante UART dans processor expert et en cliquant sur l'onglet Initialization.

9. On appelle la fonction initialize de l'objet Network et on lui passe en paramètre la structure de configuration du xbee de l'étape 7.
10. On déclare et initialise une variable du type IsnClient, le premier paramètre du constructeur est un pointeur vers l'objet network, le deuxième est le type de nœud que représente le client comme défini à la section 3.4.3.1.
11. On déclare une variable du type du capteur.
12. On passe l'adresse de la variable capteur à la fonction setSensor() du IsnClient.

13. On écrit une boucle infinie qui ne fait qu'appeler la fonction `exec()` d'`IsnClient`. À noter que si votre capteur doit exécuter du code périodiquement vous pouvez définir une fonction `exec()` dans la définition de votre capteur et l'appeler dans cette boucle infinie.
14. N'oubliez pas de changer les paramètres de compilation dans le fichier `IsnBuildConfig.h` pour compiler le bon type de capteur.

3.5.4 Procédure de débogage

Dans cette section nous faisons part des différents trucs de débogage que nous avons appris au cours du projet.

Déboguer une seule composante à la fois

Il est souvent utile de commencer par déboguer `IsnClient` tout seul puis `IsnServeur` tout seul. Ensuite lorsqu'on est satisfait du fonctionnement des deux on peut les connecter ensemble et déboguer leurs communications. Si on essaie de brancher les deux composantes dès le départ ça devient très difficile à déboguer.

Pour déboguer une composante à la fois on se sert du programme XCTU et d'un module xbee connecté à l'ordinateur par une carte xbee explorer.



Figure 36 Xbee Explorer

On s'assure d'abord que le xbee connecté avec la carte explorer est sur le même réseau que le xbee de la composante qu'on veut déboguer. Le truc ici est de faire croire à la composante à déboguer que le xbee connecté avec la carte explorer est un `IsnClient` ou `IsnServer` en lui envoyant manuellement des trames par l'entremise de XCTU.

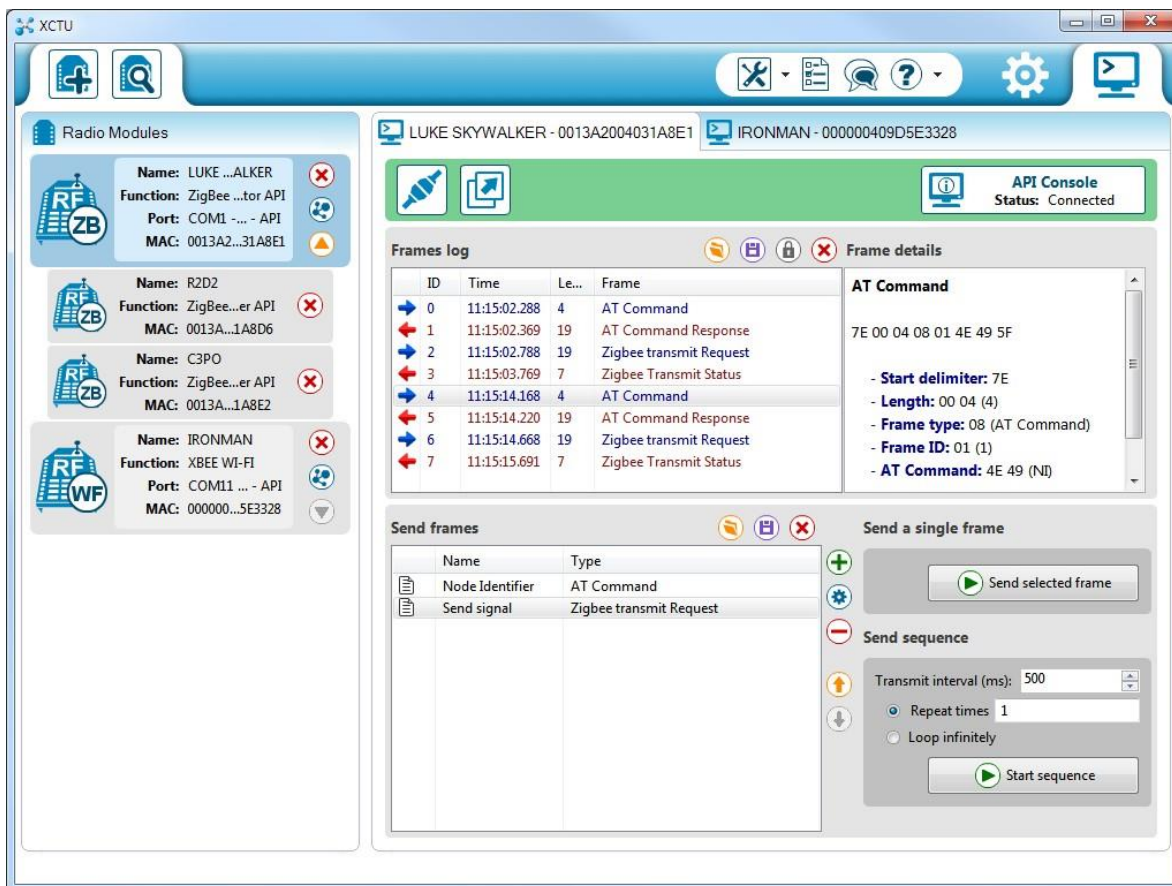


Figure 37 Interface XCTU

Pour ce faire il suffit de cliquer sur l'onglet console dans XCTU puis cliquer sur la l'icône de connexion pour établir la connexion dans le réseau. Sur la Figure 37 les flèches bleues représentent les trames reçues et les flèches rouges les trames. La fenêtre du bas montre les trames qu'on a enregistrées et qu'on peut envoyer manuellement. On peut cliquer sur l'icône avec un « + » vert pour ajouter une trame.

XBee API Frames Generator

This tool allows you to generate any kind of API frame and copy its value. Just fill in the required fields.

Protocol: **DigiMesh** Mode: **API 1 - API Mode Without Escapes**

Frame type: **0x10 - Transmit Request**

Frame parameters:

- Frame type: 10
- Frame ID: 01
- 64-bit dest. address: 00 00 00 00 00 00 FF FF
- 16-bit dest. address: FF FE
- Broadcast radius: 00
- Options: 00
- RF data:
 - ASCII
 - HEX: 01 01
- Checksum: F1

Generated frame:

```
7E 00 10 10 01 00 00 00 00 00 00 FF FF FF FE 00 00 01 01 F1
```

Byte count: 20

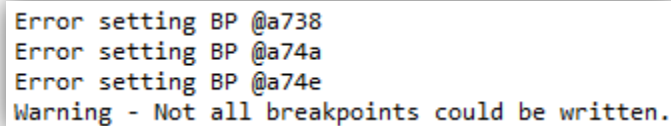
Copy frame **Close**

Figure 38 Création d'une trame

La Figure 38 illustre la fenêtre de création d'une trame. Il faut choisir le protocole DigiMesh et le « frame type » 0x10 pour transmit request. Il est mieux d'envoyer en diffusion pour le débogage pour éviter de toujours devoir changer l'adresse du récepteur. Pour ce faire on met les deux derniers octets de l'adresse 64 bits à FF FF et on laisse les autres à 0. Dans la section RF data on écrit le contenu de la trame iSN à envoyer en hexadécimal.

Erreurs de breakpoints

En déboguant avec Kinetis Design Studio il se peut que vous ayez des erreurs comme sur la Figure 39.

A screenshot of a console window showing four lines of text: 'Error setting BP @a738', 'Error setting BP @a74a', 'Error setting BP @a74e', and 'Warning - Not all breakpoints could be written.' The text is in a monospaced font with some color coding (blue for error, red for warning).

```
Error setting BP @a738
Error setting BP @a74a
Error setting BP @a74e
Warning - Not all breakpoints could be written.
```

Figure 39 Message erreur breakpoint

L'interface de débogage OpenSDA ne supporte seulement que 3 ou 4 breakpoints il faut donc aller dans le menu run de KDS et cliquer sur l'option « Remove All Breakpoints » ensuite vous pouvez ajouter un ou deux breakpoints à la fois sans erreur.

Problèmes avec les chronomètres

Si vous avez des problèmes avec les chronomètres assurez-vous d'abord que vous avez ajouté un composant processeur expert fsl_pit à votre projet. Ensuite ouvrez la configuration de ce composant et assurez-vous premièrement que la période est configurée à 1ms et que PIT IRQ handler est régler à generate code dans l'onglet events.

Deuxièmement, assurez-vous d'avoir le fichier Clock.c et Clock.h quelque part dans votre projet. Le fichier clock.c devrait ressembler à la Figure 40.

```

#include "Clock.h"

static volatile uint64_t Clock_currentMs = 1;

void Clock_resetMsCounter()
{
    Clock_currentMs = 1;
}

void Clock_incrementMsCounter()
{
    Clock_currentMs++;
}

uint64_t Clock_getMsCount()
{
    return Clock_currentMs;
}

```

Figure 40 Fichier clock.c

Finalement, assurez-vous d'avoir le code de la Figure 41 dans votre fichier Events.c.

```

void PIT_IRQHandler(void)
{
    /* Clear interrupt flag.*/
    PIT_HAL_ClearIntFlag(g_pitBase[PIT_Timer0_IDX], PIT_Timer0_CHANNEL);
    /* Write your code here ... */
    Clock_incrementMsCounter();
}

```

Figure 41 Interruption Horloge

Utilisation de la console de débogage

Nous avons codé toute une plateforme de débogage pour voir les trames iSN envoyées et reçues par IsnClient et IsnServeur. Il est aussi possible de visionner les changements d'états et les calculs de moyennes.

Pour accéder à cette fonctionnalité il faut compiler le code en dé-commentant la ligne « #define ISN_DEBUG » du fichier IsnBuildConfig.h.

Assurez-vous aussi de compiler avec l'option float dans printf. Par défaut l'éditeur de lien C++ désactive l'affiche de float avec printf car ça ajoute beaucoup de code au projet. Il est

recommandé d'activer cette option pour le débogage et de la désactiver lorsque le code est prêt à être déployé.

Pour activer cette option cliquez sur « properties » dans le menu « Project » dans KDS. Ensuite cliquez sur settings sous la section C/C++ Build dans le menu à gauche de la fenêtre. Sélectionnez ensuite « Miscellaneous » dans la section « Cross ARM C++ linker » puis cochez la case « use float with nano printf ».

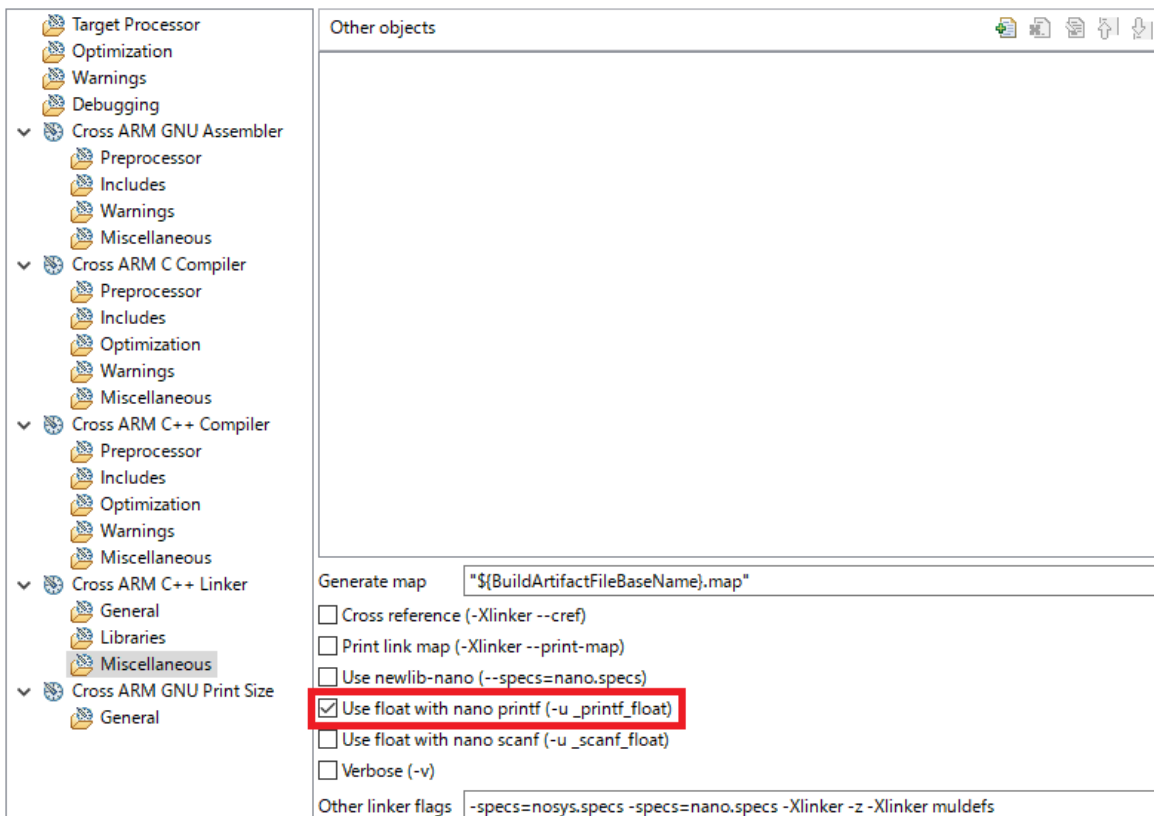


Figure 42 Option float

Ensuite vous allez devoir télécharger un programme de connexion à un port série. Nous recommandons CoolTerm qui est totalement gratuit. Pour ouvrir la connexion il faut spécifier le baudrate, vous pouvez trouver cette valeur en regardant la configuration de la composante « fsl_debug_console » de votre projet. Ensuite vous devez choisir le port série sur lequel vous connecter. Vous pouvez trouver cette valeur en ouvrant le gestionnaire de périphériques windows et en regardant la section Port COM et LPT.



Figure 43 Port COM

Vous devriez y trouver quelque chose de semblable à la Figure 43. Le port avec l'étiquette OpenSDA est normalement le bon port.

```
(00 A2 13 00 6D DD 03 41 ) ISN_MSG_MEASURE(23.94) -> IsnServer  
(00 A2 13 00 E1 23 E8 40 ) ISN_MSG_MEASURE(23.88) -> IsnServer  
ISN_SERVERSTATE_IDLE -> ISN_SERVERSTATE_SEND_MEASURE  
computeAverage() 0.00 + 23.94 + 23.88 = 47.81 / 2 = 23.91  
ISN_SERVERSTATE_SEND_MEASURE -> ISN_SERVERSTATE_IDLE
```

Figure 44 Débogage Console

Si la connexion est effectuée correctement vous devriez voir des informations apparaître dans la console de CoolTerm tel que ceux affichés à la Figure 44.

3.5.5 Choses à améliorer

Malgré le travail effectué au cours de notre projet il reste encore beaucoup de pain sur la planche afin de mener le projet à terme. Nous énumérons ici quelques points à améliorer ou à terminer dans le projet.

Publication de mesures dans un topic MQTT-SN

Dans notre implémentation, la publication des mesures dans un topic MQTT est une tâche qui relève de IsnServer. Nous croyons toutefois que cette responsabilité dépasse le cadre du protocole iSN. iSN devrait se préoccuper de l'envoi et la réception de données entre un capteur et un sink seulement. Il ne devrait en principe même pas être conscient de l'existence du client MQTT-SN. Nous croyons donc qu'un reformatage du code s'impose pour que la publication des mesures devienne une tâche spécifique au code du sink et non pas du IsnServeur.

Implémentation d'iSN pour les actionneurs

Nous n'avons pas eu le temps d'implémenter les fonctionnalités d'iSN spécifiques aux actionneurs il faudra donc que quelqu'un finisse ce travail.

Instabilité lors de l'envoi de paramètres de configuration

L'envoi de paramètres de configuration par MQTT a été implémenter mais ne fonctionne pas encore très bien. La fonctionnalité semblait bien fonctionner en modifiant un paramètre à la fois mais ne semble pas bien fonctionner lorsque plusieurs paramètres sont envoyés en même temps.

4. Apprentissages

4.1 Programmation embarquée C/C++

Nous avons appris comment faire de la programmation embarquée avec C++. Nous avons compris que les différentes broches de la carte correspondent à différents composants. Nous avons aussi appris à programmer de manière différente de manière à économiser le plus d'espace possible sur le peu de mémoire qui nous est fournie.

4.2 MQTT/MQTT-SN

Nous avons appris ce qu'est le protocole MQTT. Comment on peut utiliser un client pour se connecter à un broker, s'inscrire à des topics, publier dans des topics etc. Nous avons aussi étudié les différentes trames MQTT.

4.3 Kinetis design studio

Nous avons appris comment installer Kinetis Design Studio et le configurer pour la programmation embarquée. Nous avons vu comment créer un projet pour notre type de carte, comment déboguer, comment utiliser la fonctionnalité processeur expert. Nous avons aussi vu comment régler les zones de mémoire sur les cartes microcontrôleur et comment régler les paramètres de compilation.

4.4 XCTU

Nous avons appris à utiliser le programme XCTU de la compagnie Digi afin de changer les paramètres de configuration des xbee. Nous l'avons aussi utilisé à des fins de débogage en envoyant manuellement des trames.

4.5 Branchements

Nous avons appris comment faire des branchements sur la carte FRDMKL26Z.

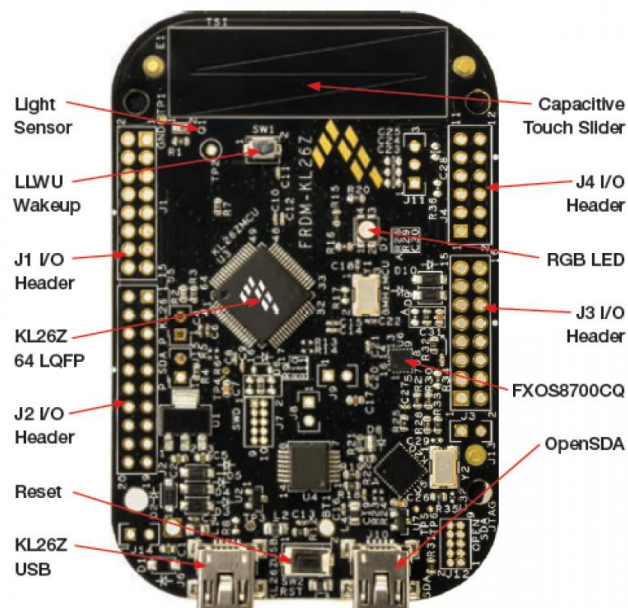


Figure 45 Pinout FRDMKL26Z

Nous avons vu qu'il y a trois UART accessible par les « JX I/O Headers » qu'on voit sur la Figure 45.

J1 02	D0	PTA1	23	TSI0_CH2	PTA1	UART0_RX
J1 04	D1	PTA2	24	TSI0_CH3	PTA2	UART0_TX

Figure 46 Pinout diagramme

Nous avons appris comment faire correspondre les différentes broches avec leur fonctionnalité. Par exemple : Si on veut brancher un xbee sur le UART0 on peut regarder la Figure 46 on conclue que la broche de réception du xbee doit être branché à la broche de transmission du UART ou UART0_TX sur la figure, on voit ensuite que UART0_TX correspond à la 4^{ième} broche de J1 sur la Figure 45.

5. Défis et difficultés rencontrées

5.1 Apprentissage

Nous avons dû faire beaucoup d'apprentissage avant même de commencer à concevoir et coder.

- Apprentissage du protocole MQTT.
- Apprentissage de la programmation embarquée.
- Compréhension du code existant, architecture etc.
- Apprentissage du branchement des composants.

5.2 Codage avec ressources limitées

Nous avons dû coder avec une mémoire limitée sur un microcontrôleur (128 KB). Gestion manuelle de la mémoire avec C++. Nous avons eu à faire très peu de programmation C++ durant notre bac. Lors du codage, nous avons rencontré un problème côté débogage; nos capacités étaient limitées avec un maximum de 3 points d'arrêt (Breakpoints). Suivit par l'implémentation de nos propres fonctions standard pour économiser l'espace de code (Ex: manipulation des chaînes de caractères).

5.3 Problèmes de connexion entre la passerelle et le broker

Nous avons noté plusieurs problèmes concernant la connexion entre la passerelle et le broker.

Une fois la connexion établie entre ces deux entités, après un certain temps, la passerelle finie par s'arrêter toute seule. Cela est peut-être dû à des fuites de mémoire. Cela reste à vérifier.

Quand on essaye de se reconnecter après une déconnexion abrupte le sink essaye de se connecter au broker via la passerelle pendant un bon 2-3 minute sans succès puis la passerelle répond « **Can't Xmit to broker** » et après la connexion fonctionne. Il s'agit là du meilleur des cas.

Dans le pire cas, la connexion du sink au broker ne fonctionne pas du tout la passerelle ne fais que répéter « **Can't recv from broker** » et « **can't xmit to broker** » tout ceci de manière indéfini

6. Conclusion

Ce projet nous a permis de mettre en pratique ce que nous avons appris au cours du bac.

- Génie logiciel;
- Réseaux;
- Programmation orientée objet;
- Développement de systèmes informatique.

Nous avons aussi appris de nouvelles choses:

- Programmation embarquée avec C++.

Le projet iSerre est un projet open source disponible sur github. Toute personne peut reprendre le projet comme nous l'avons-nous même repris.

Une fois complété, iSerre sera autant un outil de recherche indispensable qu'une solution efficace pour les horticulteurs.

Glossaire

Carte Freescale : La carte microcontrôleur (FRDMKL26Z) utilisé pour implémenter les capteurs et les « sink ».

DigiMesh : Protocole réseau propriétaire développé par la compagnie Digi. Il est utilisé dans la serre pour la communication entre les xbee.

iSN : iSerre Sensor Network, protocole application utilisé pour communiquer entre les capteurs et les « sink ».

Keepalive : Procédure utilisée pour vérifier que la connexion est toujours établie entre deux périphériques.

MQTT-SN : MQTT pour réseau de capteurs. Protocole application utilisant un modèle « Publish-Subscribe ». Dans la serre un client peut par exemple s'inscrire au « topic » température pour obtenir la température dans la serre.

Sink : Microcontrôleur équipé de deux modules radio xbee. Communique avec les capteurs pour recueillir les mesures et avec la passerelle pour publier les mesures. Peut aussi servir de point de contrôle pour envoyer des commandes aux actionneurs.

UART : universal asynchronous receiver/transmitter, c'est une composante matériel permettant de faire des entrées et sortie de données en série. La carte freescale (FRDMKL26Z) en compte trois dont un est utilisé pour fournir la console de débogage. Dans notre projet les deux autres sont utilisés pour communiquer avec les xbee, les capteurs et les actionneurs.

Xbee : Appareil émetteur récepteur radio.

ZigBee : Protocole réseau basé sur la spécification IEEE 802.15.4 pour les petits émetteur-récepteur radios.

Annexe

Lien github vers le code :

<https://github.com/VincentEmond/iSerre-1>

Lien vers la spécification de MQTT-SN :

http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf

Lien de téléchargement pour CoolTerm :

<http://freeware.the-meiers.org/>