Vayel

Zeste de savoir

Introduction au protocole WAMP

12 août 2019

Table des matières

Figure 0. – Logo de WAMP

WAMP, pour *Web Application Messaging Protocol*, est un **protocole**[1] open source basé sur WebSocket permettant de faire communiquer des pairs **découplés** en **temps réel**[2]. Il sera question dans ce tutoriel d'introduire les concepts sous-jacents à ce protocole. Mais avant cela, parlons un peu des fameuses websockets et de la quête du temps réel.

Seule une culture générale en Web est requise pour suivre ce tutoriel : tout le vocabulaire nécessaire à sa compréhension sera défini. Néanmoins, nous ne reviendrons pas sur tous les termes propres au domaine du Web : si vous en rencontrez un qui vous est inconnu, il n'est probablement pas primordial pour comprendre l'ensemble du texte, et une recherche rapide vous permettra de vous renseigner puis de poursuivre la lecture en toute sérénité. Sachez également que ce tutoriel se contente d'*introduire* les *concepts* relatifs à WAMP : vous n'apprendrez pas ici à utiliser les bibliothèques connexes à ce protocole, ni n'étudierez sa spécification technique.

1. Et non la suite d'outils pour déployer son site Web sous Windows.
2. Plus précisément, il est question ici de ce qu'on appelle le temps réel mou .

1. Le protocole WebSocket

Aujourd'hui, le protocole HTTP est massivement employé pour consulter une page Web. Normal, il a été conçu pour : son objectif est de délivrer du texte. Sauf qu'à sa création, on n'imaginait pas qu'Internet prendrait autant d'ampleur, et les possibilités d'interaction dans une page étaient très pauvres : vous deviez, pour mettre à jour le moindre élément, charger derechef la page dans son intégralité. Eh oui, dans son intégralité. Autant dire que ça pompait au niveau du réseau et des ressources du serveur, et ne parlons même pas du confort pour l'utilisateur : on était loin de la communication en temps réel.

Fort heureusement, AJAX est paru. Cette technologie permet d'effectuer via JavaScript une requête HTTP sans rechargement de la page. Il est alors possible de mettre à jour une partie seulement de cette dernière à partir de données provenant du serveur et donc, en contactant fréquemment ce dernier, de simuler du temps réel. Mais aussi pratique qu'AJAX puisse être, les requêtes se font toujours via HTTP. Or ce protocole est très limité pour faire du temps réel : il est *sans état* et ne permet pas le *push*.

« Sans état » signifie qu'à la fin de toute requête, la connexion avec le client est fermée. L'avantage est que le serveur ne s'embête pas à retenir des informations sur les clients ni à identifier à quelle session correspond une requête ; il peut de ce fait traiter un grand nombre de connexions. Seulement, communiquer en temps réel en utilisant HTTP contraint à ouvrir et fermer le canal de communication à de nombreuses reprises. On comprend sans peine que c'est loin d'être optimal, surtout dans le cas d'échanges chiffrés.

Le terme *push* désigne pour un serveur Web le fait de contacter un client de son plein gré. Avec HTTP, le premier ne peut que répondre au second, c'est-à-dire attendre qu'on le contacte pour pouvoir envoyer des données. Logique, me direz-vous, HTTP n'a été conçu que pour délivrer une page Web demandée. Sauf qu'aujourd'hui on veut du temps réel ; le client est alors obligé d'interroger le serveur très fréquemment pour obtenir les informations. Là encore, ce n'est pas l'idéal.

Le protocole WebSocket[3], basé sur l'architecture client-serveur à l'instar de HTTP, cherche à pallier ces limitations. Il permet d'établir une connexion *full-duplex persistente* entre un client et un serveur. « *Full-duplex* », ça veut dire que les deux pairs peuvent se contacter l'un l'autre, de leur propre chef et simultanément. Notamment, le serveur peut désormais notifier le client d'une information à n'importe quel moment. « Persistente », parce que la connexion n'est pas fermée à la fin d'une requête, contrairement à HTTP. On imagine volontiers le gain de performances.

2. Des messages et des composants

WAMP se veut une surcouche de WebSocket et permet donc de faire communiquer des pairs en temps réel. Ceux-ci, appelés « composants » (en anglais : *components*), s'échangent des messages au travers du réseau par le biais d'un programme qu'on appelle un routeur. Concrètement, ce dernier est en premier lieu exécuté[4], puis chacun des composants est démarré, s'y connecte et communique avec les autres à travers lui.

3. L'expression « les websockets », bien que très usitée, est un abus de langage.

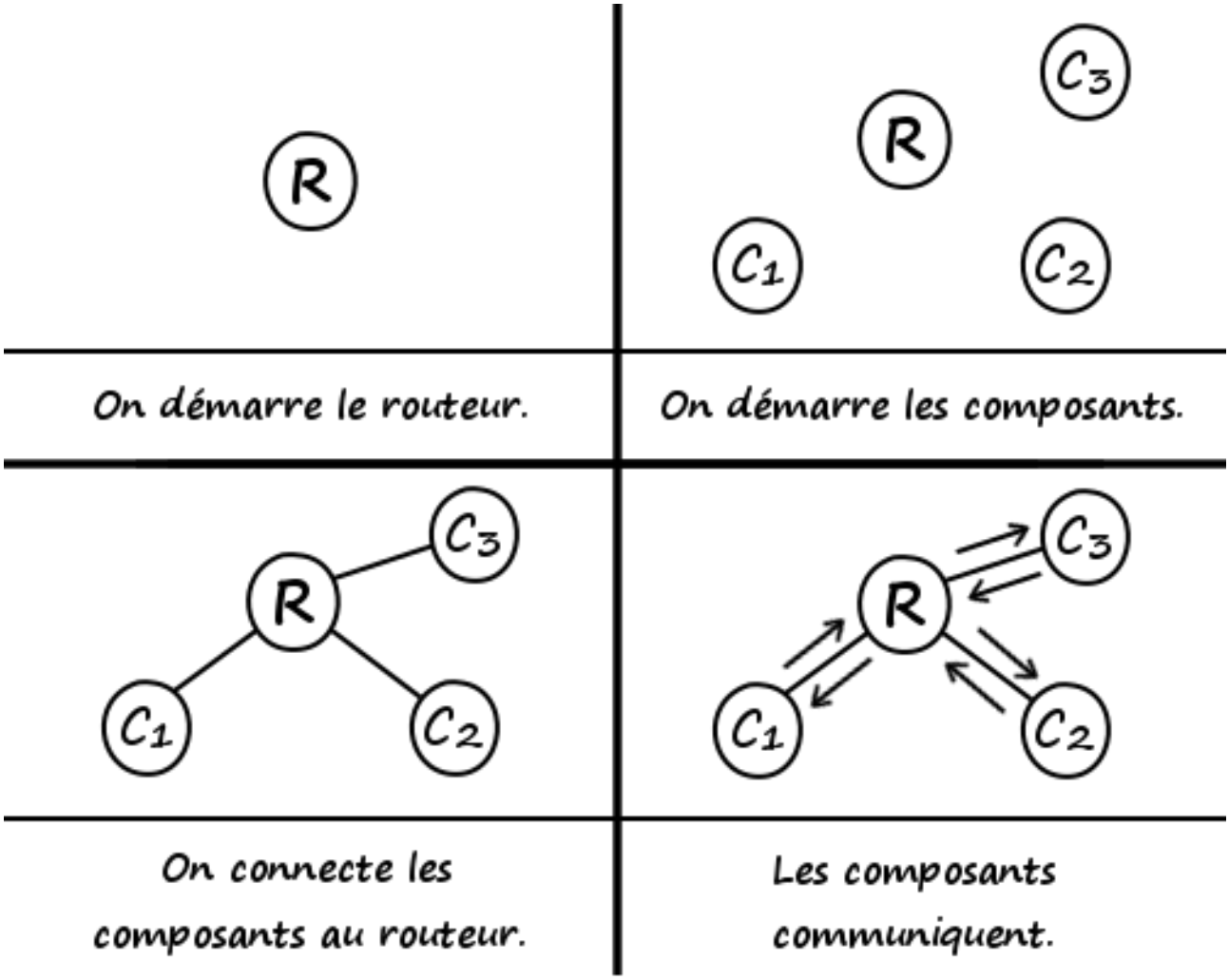

FIGURE 2. – Un routeur pour les connecter tous, et par des messages les relier.

Cela permet d'obtenir un système très souple, avec des pairs ne se connaissant pas et s'exécutant possiblement sur plusieurs machines. En effet, pour joindre tous les autres, un composant a uniquement besoin de pouvoir contacter le routeur : il se retrouve donc complètement découplé du reste de l'application, laquelle est ainsi constituée de plusieurs modules indépendants.

? Mais comment fait un composant pour contacter les autres tout en ignorant leur existence ?

En réalité, il ne le fait pas. Un pair ne peut qu'envoyer un message, lequel sera réceptionné par le routeur puis possiblement transmis à un ou plusieurs composants. Seul le routeur a donc connaissance de tout le monde ; les pairs, eux, se contentent d'appeler un service ou de divulguer une information en lui expédiant des messages.

Ces derniers sont composés de deux parties : un intitulé (en anglais : *topic*) sous forme d'une chaîne de caractères, auquel on joint des données, converties au format JSON[6]. Par exemple :

```
sendMessage("createCharacter", {
    "name": "Salim",
    "surname": "Condo",
    "gender": 0,
    "inventory": [
        {
            "type": "corde",
            "name": "Fil d'Hulm",
            "quantity": 1
        }
    ]
})
```

L'avantage d'un tel système est que tout composant peut être écrit dans n'importe quel langage, pour peu qu'il existe une bibliothèque permettant d'expédier des messages au routeur. Aujourd'hui, des implémentations sont disponibles pour Python, Node.js, C++, Erlang, PHP, C#, Java, Tessel, et même pour les clients Web, vu que le protocole repose sur WebSocket[5]. Plus rien alors ne vous empêche de déployer des architectures telles que les suivantes, où le routeur n'est pas représenté.[7]

— **Un jeu par navigateur**

Les clients utilisent des navigateurs en guise d'interfaces et un composant écrit en Python s'occupe de gèrer le jeu : tel joueur peut-il se déplacer ici ? A-t-il assez d'argent pour acheter cela ? Etc. Il faut comprendre que le composant Python n'est pas nécessairement un serveur Web : il se charge uniquement de la gestion du jeu.

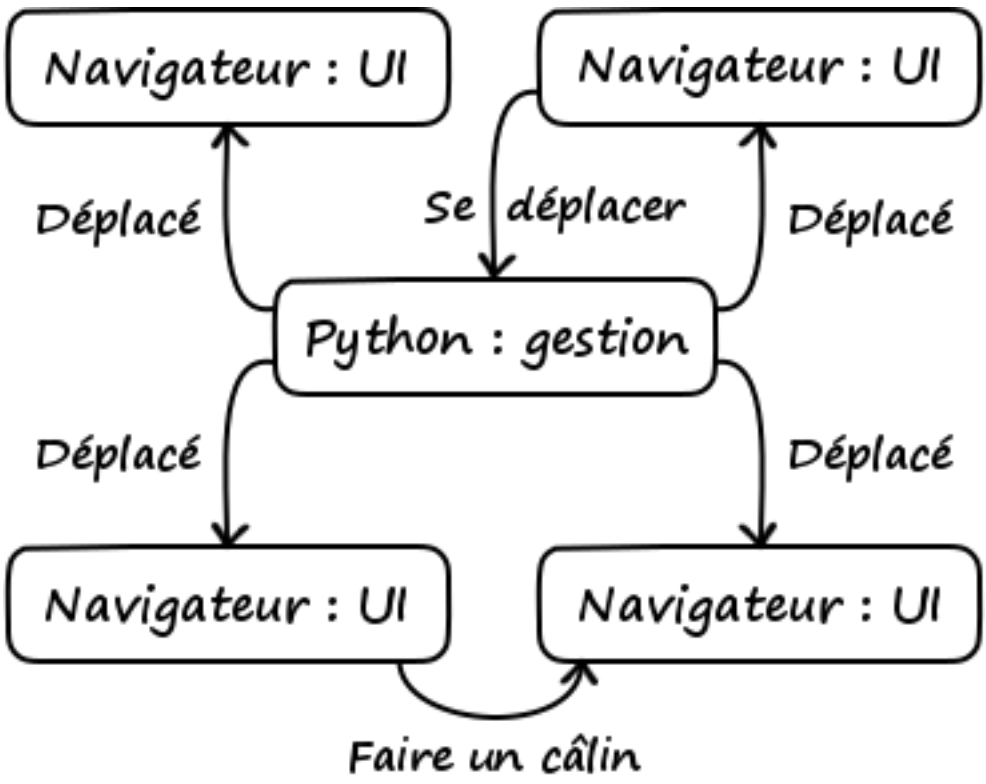

FIGURE 2. – Un jeu par navigateur.

Le composant en haut à droite demande ici au composant Python s'il peut se déplacer et, le cas échéant, le second notifie tout le monde du déplacement. Il est également possible de faire communiquer deux clients Web entre eux, comme le font les deux composants du bas.

On constate qu'en l'absence de routeur, la complexité des connexions entre les composants augmente rapidement et que chacun devient fortement couplé aux autres. Imaginez ce que deviendrait le schéma si tous les navigateurs voulaient faire un câlin à chacun des autres.

— **Un peu de domotique**

L'Arduino Yun peut faire office de composant WAMP. Il est alors simple, avec un routeur, de connecter la carte électronique à n'importe quel composant en guise d'interface : application Android, application Web, de bureau...

FIGURE 2. – Un peu de domotique.

WAMP permet ici de disposer de plusieurs interfaces sans rien changer au code du composant en charge de l'acquisition et tout est complètement découplé. En effet, si jamais un jour vous souhaitez remplacer votre Arduino par, mettons, une RaspberryPi, vous n'aurez pas besoin de retoucher le code des autres composants, puisque pour eux, récupérer la température se fera toujours en envoyant un même message, quel que soit le composant chargé de la mesure. En outre, vous pouvez ajouter ou enlever comme bon vous semble un composant d'interface, sans interférer avec le reste.

— **Traitement de verger**

Pour moduler la dose de produit de traitement dans un verger, votre programme localise le tracteur dans ce dernier puis regarde sur une carte (établie auparavant) quelle quantité de fruits comporte l'arbre considéré. Il en déduit alors la dose de produit à distribuer et la distribue.

Pour plus de flexibilité, un tel programme ne devrait pas faire d'hypothèse sur le moyen d'acquisition de la position du tracteur, ni sur la méthode de calcul de la dose à partir de la quantité de fruits. Avec WAMP, il est possible de partager votre application en plusieurs modules indépendants (localisation, distribution, etc.) communiquant entre eux par des messages. À condition de respecter le format de ces derniers, tout agriculteur pourra alors implémenter son propre module de localisation, par exemple, et ce sans modifier le reste de l'application.

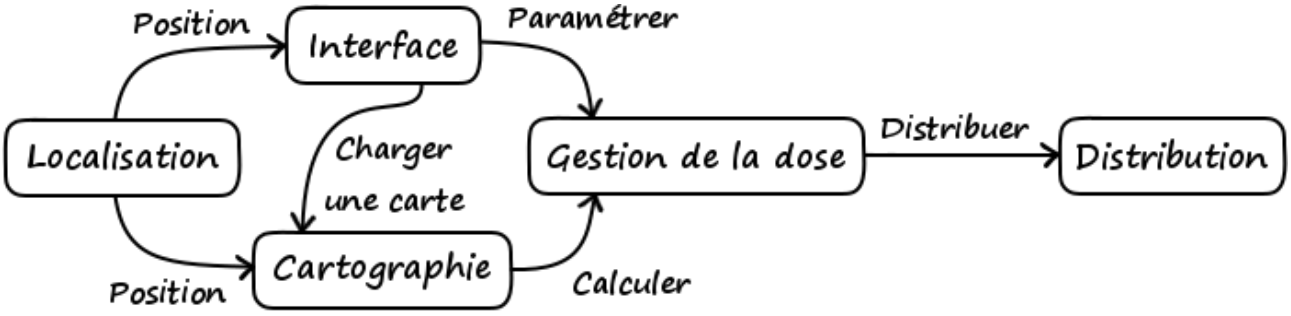

FIGURE 2. – Traitement de verger.

Ici, le module de localisation notifie les autres de la position à intervalles réguliers. À la réception d'une telle information, le module de cartographie en déduit la quantité de fruits de l'arbre traité puis secoue le module de gestion de la dose pour qu'il calcule celle à distribuer et demande à la carte électronique de le faire. Le module d'interface permet d'afficher la position et de paramétrer les autres composants.

Il est temps maintenant de nous attarder un peu plus sur les messages, et notamment sur la manière dont ils sont expédiés. Ou plutôt, les manières : WAMP en rassemble deux dans un même protocole.

3. Remote Procedure Call

La première méthode pour expédier des messages est appelée *Remote Procedure Call* (RPC). Sans surprise, elle permet d'appeler une procédure définie par un autre composant puis d'en récupérer le résultat.

3.1. Un cas de figure

Considérons l'exemple d'architecture suivant.

http://zestedesavoir.com/media/galleries/1508/

FIGURE 3.

4. Au sens informatique du terme, n'ayez crainte.
5. En fait, WebSocket n'est pas nécessaire à l'implémentation de WAMP. Plus ici .
6. C'est pourquoi certains objets ne pourront être expédiés sans passer par la sérialisation.
7. Des exemples plus généraux ici .

Pour simplifier, supposons qu'il n'y a que deux composants : un sur l'Arduino et un sur Android, le second souhaitant demander la température extérieure au premier. Dans une version simpliste, il effectuerait un appel RPC vers l'Arduino en contactant la carte électronique, laquelle, à la réception de cette commande, s'exécuterait et retournerait le résultat mesuré. Sauf qu'un tel fonctionnement ne prend pas en compte le fait que deux pairs ne se connaissent pas, et l'appel va en réalité se dérouler autrement :

Le composant démarré sur l'Arduino se connecte au routeur puis indique à ce dernier qu'il souhaite relier une de ses procédures, la méthode `get_temperature`, à l'intitulé « com.app.gettemp[9] ». Lorsque le composant Android souhaite s'enquérir de la température, il envoie le message « com.app.gettemp » au routeur, lequel le transmet au composant Arduino, qui exécute la méthode `get_temperature` et lui retourne le résultat. Le routeur fait finalement parvenir ce dernier au composant Android, qui, à aucun moment, n'a su où se situait la méthode permettant de récupérer la température extérieure.

Résumons ce processus par des schémas. Une ligne pleine indique une connexion au routeur, une en pointillés une connexion éventuelle et une ligne fléchée désigne l'envoi d'un message.

FIGURE 3. – Le composant Arduino demande au routeur de rattacher l'intitulé « com.app.gettemp » à sa procédure get_temperature.

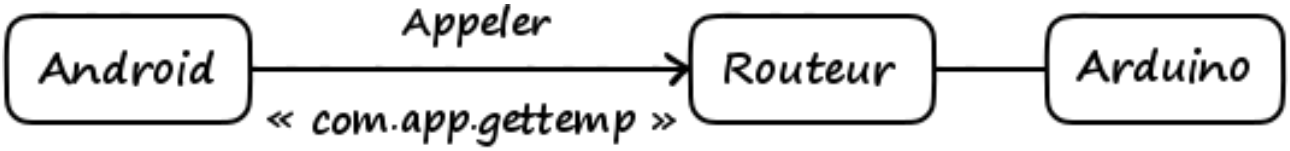

FIGURE 3. – Le composant Android demande au routeur d'appeler la procédure rattachée à l'intitulé « com.app.gettemp ».

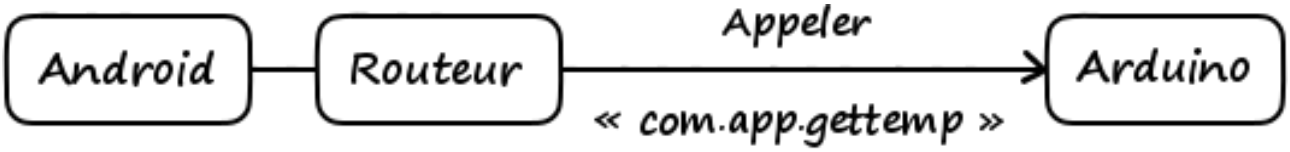

FIGURE 3. – Le routeur transmet l'appel au composant hébergeant ladite procédure.

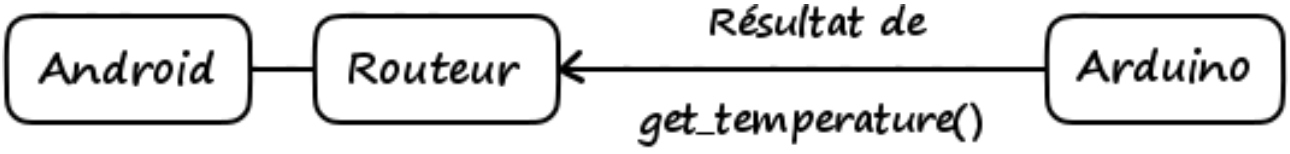

FIGURE 3. – Le composant Arduino retourne au routeur le résultat généré par l'exécution de la procédure.

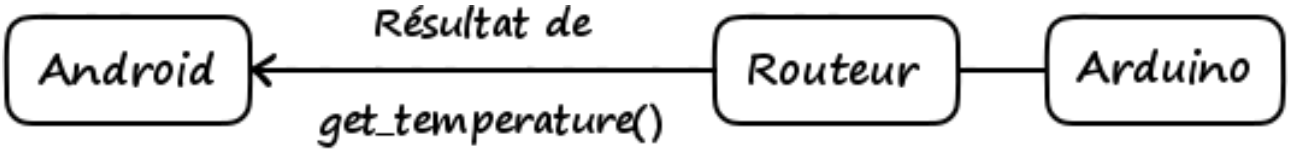

FIGURE 3. – Le routeur transmet ce résultat au composant ayant appelé la procédure.

Ici, aucune donnée n'est jointe au message, simplement parce que la procédure ne prend pas de paramètres. Mais on aurait pu indiquer la précision souhaitée dans la mesure, ou que sais-je d'autre. Évidemment, la présence et la nature des données dépendent des paramètres requis par la procédure appelée.

3.2. Plus généralement

Lors d'un échange via RPC, deux composants interviennent : l'appelant (en anglais : *caller*) et l'appelé (en anglais : *callee*). Le premier souhaite exécuter (en anglais : *call*) une procédure rattachée à un intitulé particulier. Il effectue alors sa demande en envoyant ledit intitulé ainsi que des données au routeur[10], lequel les communique au composant ayant enregistré (en anglais : *register*) la procédure. Ce composant, l'appelé, exécute cette dernière avec les paramètres renseignés et retourne un résultat au routeur, qui le transmet pour terminer à l'appelant.

Il est important de noter que l'appelant et l'appelé n'ont à aucun moment su qui était l'autre : le premier a ignoré tout du long où se situait la procédure, et le second quel composant l'appelait.

Tout se déroule à merveille dans ce cas fictif, mais il existe tout de même deux possibilités d'erreur : l'exécution de la procédure en génère une, ou bien l'intitulé demandé n'a été rattaché à une procédure par aucun composant.

Notons qu'il est possible à l'appelé de ne rien retourner. Par exemple, un appel RPC pourrait simplement demander une insertion dans une base de données. Le cas échéant, il n'y aurait en principe pas de valeur de retour, seulement une erreur éventuelle lors de l'exécution de la procédure.

De plus, en pratique, l'appel s'effectue de manière asynchrone : une fois le message envoyé, l'appelant est libre de faire ce qu'il veut ; il sera simplement notifié par le routeur lorsque le résultat sera prêt. Autrement dit, l'appel RPC ne bloque pas l'appelant, lequel ne perd pas son temps à attendre que l'appelé accomplisse son travail.[8] Heureusement me direz-vous : on n'imaginerait pas rester devant sa boîte aux lettres en attendant de recevoir un retour de son correspondant.

Résumons tout cela par un schéma.

http://zestedesavoir.com/media/galleries/1508/

8. Parfois, on souhaite que ce soit bloquant, dans le cas d'une synchronisation par exemple. Il faut alors bidouiller en fonction de la bibliothèque et du langage utilisés.

FIGURE 3. – Illustration du fonctionnement de RPC.

3.3. D'autres exemples

FIGURE 3. – Tout ce que vous faites avec AJAX.

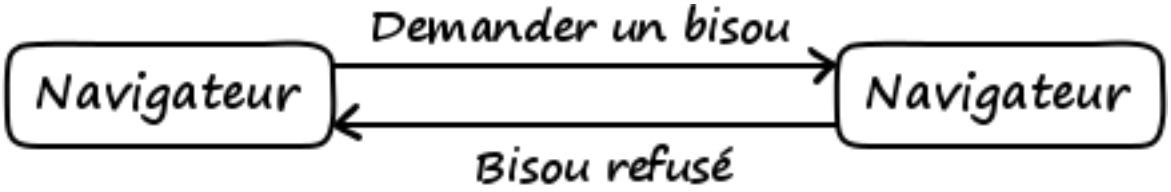

FIGURE 3. – Les navigateurs peuvent directement communiquer entre eux. Ou pas...

http://zestedesavoir.com/media/galleries/1508/

FIGURE 3. – L'appelé peut ne rien retourner d'autre qu'une erreur éventuelle.

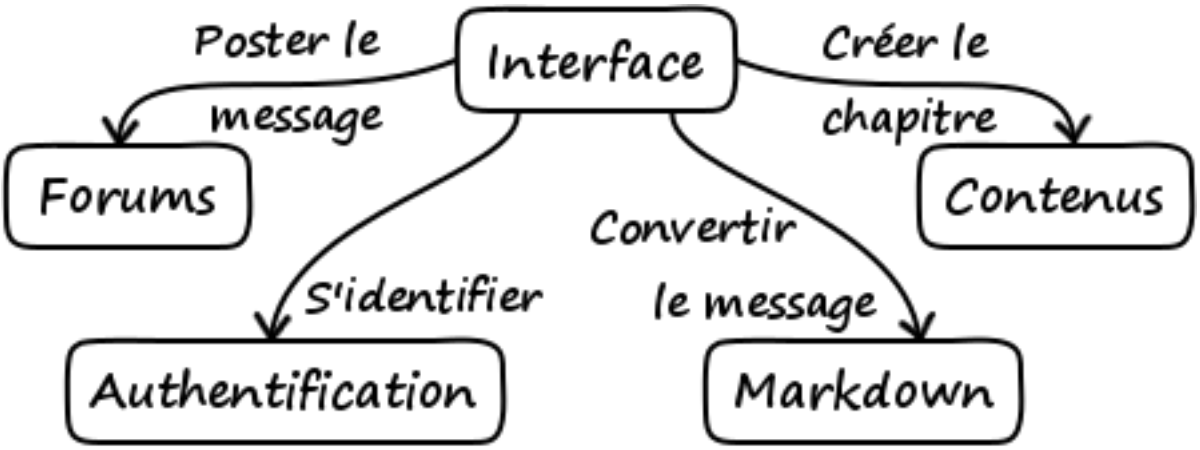

FIGURE 3. – Des micro-services : des modules atomiques que vous pouvez exécuter sur plusieurs machines, redémarrer indépendamment, écrire dans n'importe quel langage, réutiliser facilement, remplacer sans problème...

3.4. Le code

En JavaScript, avec la bibliothèque Autobahn|JS , on effectue et reçoit des appels RPC de cette manière.

```
(function() {
    var connection = new autobahn.Connection({ // AutobahnJS est un client WAMP pour JavaScript.
        url: 'wss://demo.crossbar.io/ws',      // L'url du routeur. La communication est chiffrée.
        realm: 'realm1'                        // Un namespace pour les intitulés de messages.
    });

    connection.onopen = function(session, details) {
        console.log('Connecté.'); // Poil au nez

        // Enregistrement d'une procédure
        function rpcCallback(args, kwargs) {
            console.log(args);   // [...]
            console.log(kwargs); // {...}

            if(kwargs['type'] == 'baveux') {
                return false; // Bisou refusé
            } else {
                return true;
            }
        }

        session.register('com.example.kiss', rpcCallback).then(
            function(reg) {
                console.log('Procédure enregistrée avec succès.');
            },
            function(err) {

                    console.log("La procédure n'a pu être enregistrée : ", err);
            }
        );

        // Appel à une procédure
        session.call('com.example.move', [ // args
            3, // x
            5 // y
        ], { // kwargs
            'mean': 'roulades'
        }).then(
            function(res) {
                console.log("Résultat de l'appel RPC : ", res);
```

```
            },
            function(err) {
                console.log("Erreur lors de l'appel RPC : ", err);
            }
        );
    };

    connection.onclose = function(reason, details) {
        console.log('Connexion fermée.');
    };

    connection.open();
})();
```

La procédure rattachée à l'intitulé « com.example.move » serait définie de la façon suivante avec Autobahn|Python .

```
from autobahn import wamp
from autobahn.asyncio.wamp import ApplicationSession,
    ApplicationRunner # On peut aussi utiliser Twisted


class SexyComponent(ApplicationSession):

    def __init__(self, config):
        ApplicationSession.__init__(self, config)

        self.msg = config.extra['msg']

    def onJoin(self, details):
        """Le composant est connecté au routeur : on enregistre des
        procédures.
        """

        self.register(self)

    @wamp.register('com.example.move')
    def proc(self, x, y, z=0, mean=None):

            """Procédure exécutée lors de l'appel à 'com.example.move'."""

        print(x) # 3
        print(y) # 5
        print(z) # 0
        print(mean) # 'roulades'

        return self.msg

```

```
30
31 if __name__ == '__main__':
32     ApplicationRunner(
33         url = 'wss://demo.crossbar.io/ws', # Même routeur que le
               composant JS
34         realm = 'realm1',                  # Même namespace que le
               composant JS
35         extra = {'msg': 'Mouarfarfarf !'}  # Paramètres passés au
               composant Python
36     ).run(SexyComponent)
```

4. Publish and Subscribe

RPC s'avère inapproprié pour communiquer la même information à plusieurs composants. La seconde méthode implémentée par WAMP pour envoyer des messages, *Publish and Subscribe* (PubSub), pallie ce manque. Ici, on ne cherche pas à exécuter une procédure particulière, mais à notifier certains composants d'une information.

4.1. Un exemple

Revenons sur l'exemple d'architecture suivant.

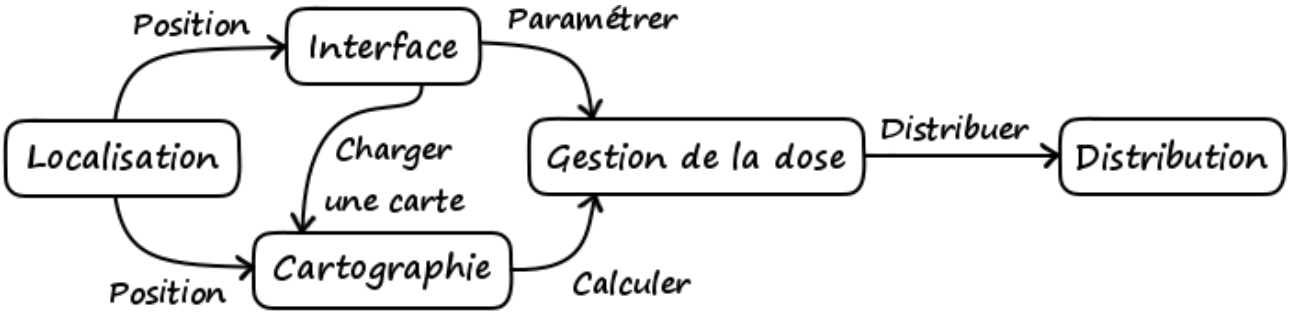

FIGURE 4.

À intervalles réguliers, le composant de localisation notifie les modules d'interface et de cartographie de la position du tracteur dans le verger : il leur envoie simplement un message « com.app.position » comportant les données de localisation.

Plus précisément, toujours dans l'idée d'assurer le découplage des composants, le module de localisation n'expédiera pas son message directement aux deux autres : il l'enverra plutôt au routeur, lequel le transmettra aux composants lui ayant au préalable indiqué être intéressés par

9. Ce format n'est pas obligatoire. Plus ici .
10. Pour RPC, on parle spécifiquement de *dealer*.

l'intitulé « com.app.position ». Dans le cas présent, ces composants sont les modules d'interface et de cartographie.

Résumons cela par des schémas. Une ligne pleine indique une connexion au routeur, une en pointillés une connexion éventuelle et une ligne fléchée désigne l'envoi d'un message.

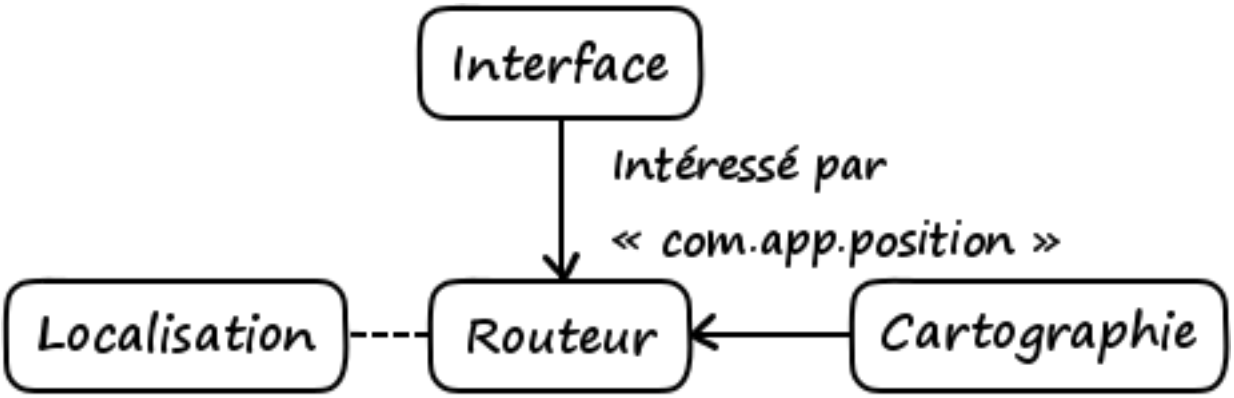

FIGURE 4. – Des composants indiquent au routeur qu'ils sont intéressés par l'intitulé « com.app.position ».

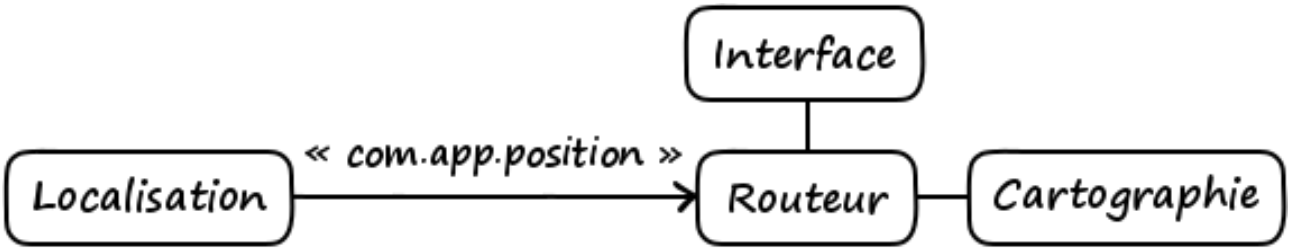

FIGURE 4. – Un composant envoie un message intitulé « com.app.position » au routeur.

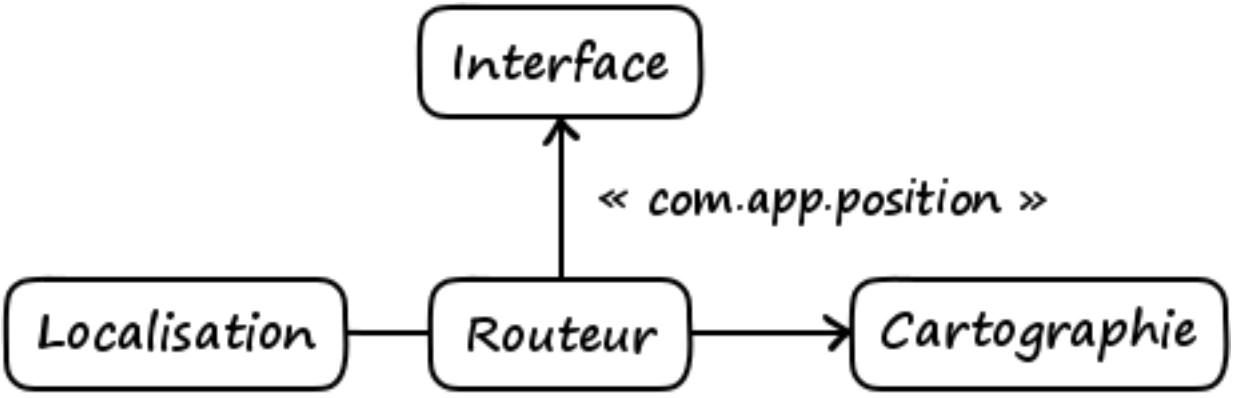

FIGURE 4. – Le routeur transmet le message aux intéressés.

4.2. Plus généralement

PubSub fait intervenir un composant appelé publieur (en anglais : *publisher*) et éventuellement d'autres pairs, les abonnés (en anglais : *subscribers*). Les seconds s'abonnent (en anglais : *subscribe*) à un *topic*, *i.e.* indiquent au routeur qu'ils souhaitent recevoir les messages portant ledit intitulé[11]. Dès lors, à chaque fois qu'un composant, le publieur, envoie un tel message au routeur[12], ce dernier le transmet à tous les intéressés, *i.e.* à tous les abonnés.

Il faut bien comprendre que ce système se base sur des évènements : les abonnés n'ont pas à attendre bêtement d'être notifiés, et heureusement. Autrement dit, ils spécifient une fonction[13] à exécuter au moment de la réception de la notification puis peuvent faire autre chose en attendant l'information.

Le publieur ignore complètement qui reçoit le message ; il se contente d'informer le routeur d'un évènement, lequel en fera part aux intéressés. Il est donc possible que l'information ne parvienne à personne, comme il est possible pour un composant de s'abonner et de se désabonner à tout moment, au nez et à la barbe des autres — excepté du routeur, bien entendu. De leur côté, les abonnés ne se connaissent pas et ignorent de quel composant provient le message.

En outre, du fait de la multiplicité potentielle des destinataires, il n'est plus possible pour le composant qui publie d'avoir de retour : contrairement à RPC, les destinataires, *i.e.* les abonnés, ne peuvent renvoyer de résultat et les erreurs qu'ils rencontrent ne sont pas transmises à l'expéditeur, *i.e.* le publieur.

Résumons cela par un schéma.

FIGURE 4. – Illustration du fonctionnement de PubSub.

4.3. D'autres cas de figure

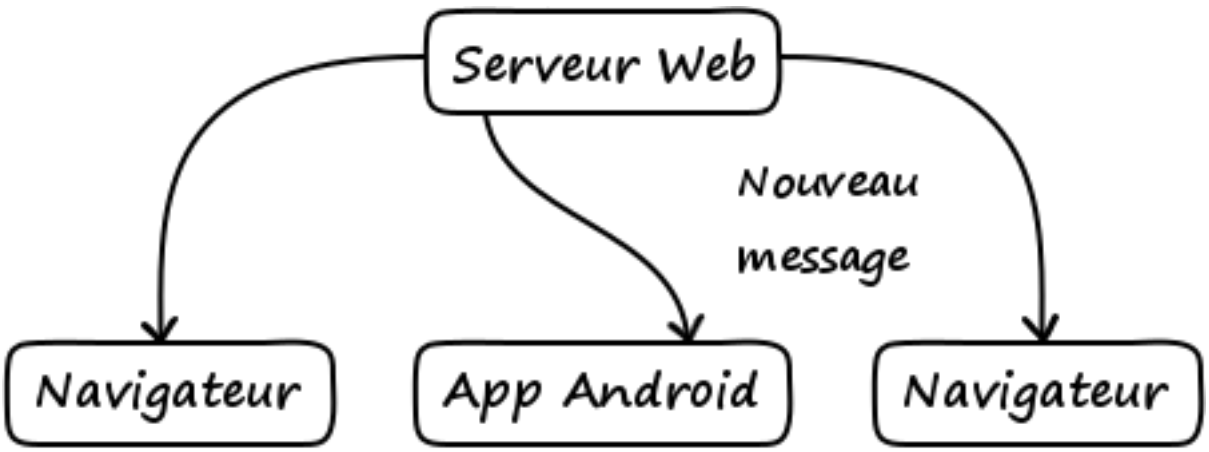

FIGURE 4. – Connecter son UI, quelle qu'elle soit, au routeur puis s'abonner permet par exemple d'être notifié en temps réel des nouveaux messages sur le forum.

FIGURE 4. – Les navigateurs peuvent directement s'envoyer des messages.

4.4. Le code

En JavaScript, on s'abonnerait et publierait des messages comme dans le code qui suit.

```
(function() {
    var name = prompt('Quel est votre nom ?');

    var connection = new autobahn.Connection({ // AutobahnJS est un client WAMP pour JavaScript.
        url: 'wss://demo.crossbar.io/ws',      // L'url du routeur. La communication est chiffrée.
        realm: 'realm1'                        // Un namespace pour les intitulés de messages.
    });

    connection.onopen = function(session, details) {
        console.log('Connecté.');

        // Abonnement à des sujets
        function posCallback(args, kwargs) {
            // Un joueur s'est déplacé
            console.log(args);   // [...]
            console.log(kwargs); // {...}
        }

        session.subscribe('com.example.position', posCallback).then(
            function(sub) {
                console.log('Abonné au topic avec succès.');
            },
            function(err) {
                console.log("Impossible de s'abonner au topic : ", err);
            }
        );

        function msgCallback(args, kwargs) {
```

```
            console.log(args[1] + ' a dit : ' + args[0]);
        }

        session.subscribe('com.example.message', msgCallback).then(
            function(sub) {},
            function(err) {}
        );

        // Publication d'un message
        session.publish('com.example.message', [ // args
            'Ouech les potos ! Bien ou bien ?',
            name
        ], {/* kwargs */}, { // options
            // Par défaut, on n'est pas notifié des messages
            // qu'on publie, même si on y est abonné. Pour
            // cela, il faut activer une option à la publication.
            acknowledge: true
        });
    };

    connection.onclose = function(reason, details) {
        console.log('Connexion perdue.'); // Poil au, euh...
    };

    connection.open();
})();
```

En Python, on s'abonnerait de la manière suivante.

```
from autobahn import wamp
from autobahn.asyncio.wamp import ApplicationSession,
    ApplicationRunner


class SexyComponent(ApplicationSession):

    def __init__(self, config):
        ApplicationSession.__init__(self, config)

        self.msg = config.extra['msg']

    def onJoin(self, details):
        """Le composant est connecté au routeur : on s'abonne."""

        self.subscribe(self)

    @wamp.subscribe('com.example.message')
    def callback(self, text):
```

```
19
            """Appelé quand on est notifié de l'intitulé 'com.example.message'
20
21          print(text) # 'Ouech les potos ! Bien ou bien ?'
22          print(self.msg) # Histoire d'utiliser les paramètres passés
                au composant
23
24
25  if __name__ == '__main__':
26      ApplicationRunner(
27          url = 'wss://demo.crossbar.io/ws', # Même routeur que le
                composant JS
28          realm = 'realm1',                  # Même namespace que le
                composant JS
29          extra = {'msg': 'Message reçu'}    # Paramètres passés au
                composant Python
30      ).run(SexyComponent)
```

4.5. Démonstration

Petite démonstration de PubSub. Ne jouez pas trop longtemps avec.

!(https ://jsfiddle.net/uoy0ytm5/8/)

En définitive, WAMP est un protocole open source basé sur WebSocket et permettant de faire communiquer en temps réel des pairs découplés, lesquels s'envoient des messages via un routeur et par le biais de deux méthodes :

— RPC, pour appeler une procédure distante ;
— PubSub, pour notifier certains composants d'une information.

Pour ceux souhaitant recueillir plus d'informations sur le sujet ou, pourquoi pas, bidouiller avec WAMP, je conseille de consulter les ressources suivantes.

— wamp.ws , le site officiel de WAMP
— Crossbar.io , un routeur pour WAMP
— Autobahn , une suite de bibliothèques pour écrire ses clients WAMP
— Le blog de Tavendo , l'entreprise allemande à l'origine de WAMP

11. Plus exactement, les messages portant cet intitulé et expédiés via PubSub.
12. On parle spécifiquement de *broker* dans le cas de PubSub.
13. Cette fonction est alors compréhensiblement appelée *callback*.

5. Crédits

Le logo de WAMP est la propriété de Tavendo GmbH.

Un grand merci à Arius, le loup le plus pulpeux de l'univers, pour la validation de ce tutoriel.

Liste des abréviations

AJAX Asynchronous JavaScript and XML. 2

HTTP HyperText Transfer Protocol. 2

JSON JavaScript Object Notation. 3

PubSub Publish and Subscribe. 12, 14, 17

RPC Remote Procedure Call. 6–8, 10, 12, 14, 17

WAMP Web Application Messaging Protocol. 1, 2, 5, 6, 12, 17, 18